Virtual Platforms for embedded system prototyping

Francesco Paterna, Martino Ruggiero, Luca Benini

DEIS – Micrel lab

francesco.paterna@unibo.it

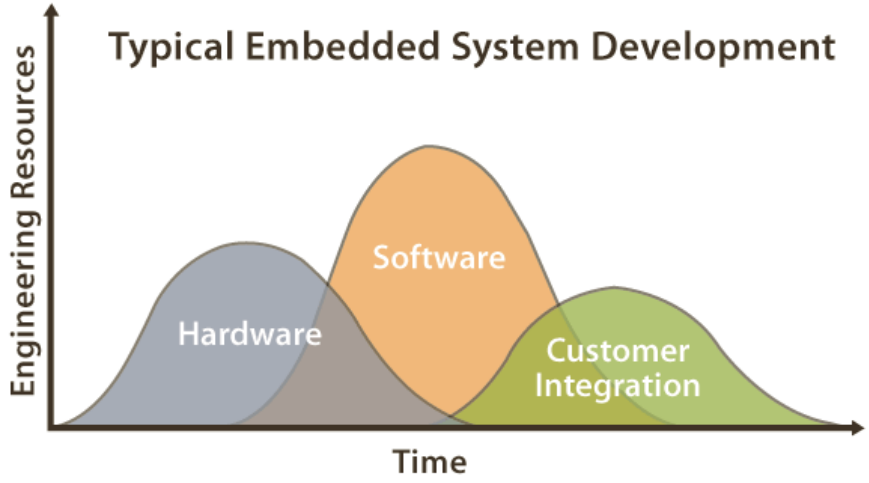
IL PRESENTE MATERIALE È RISERVATO AL PERSONALE DELL'UNIVERSITÀ DI BOLOGNA E NON PUÒ ESSERE UTILIZZATO AI TERMINI DI LEGGE DA ALTRE PERSONE O PER FINI NON ISTITUZIONALI

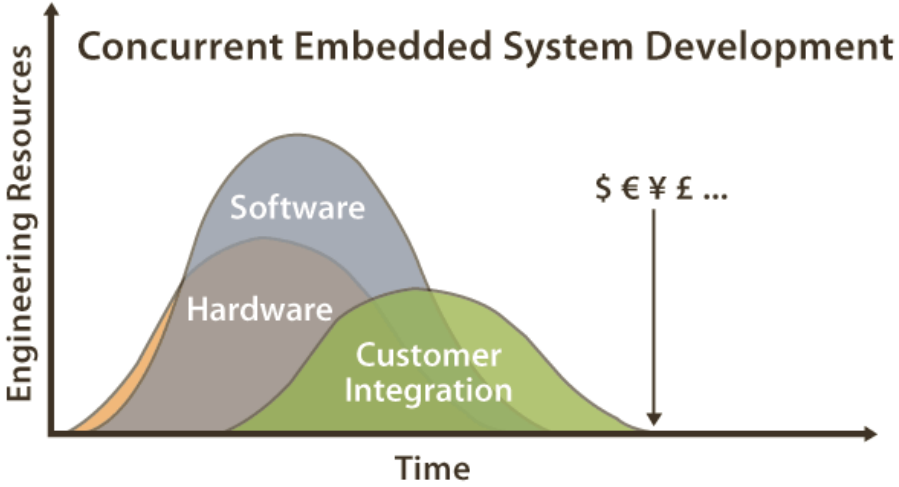
EDA Requirements and Challenges

- Typical design cycles are experiencing more and more **schedule pressure**
  - Need for a very fast product cycle in the marketplace

  Recent growth in complexity of Embedded System Designs (HW+SW) requires a fundamental new approach and methodology.
- Shortened Design Cycle → Need For **Concurrent Design**

  Simulation Technology advancements are benefiting **all phases of system development**

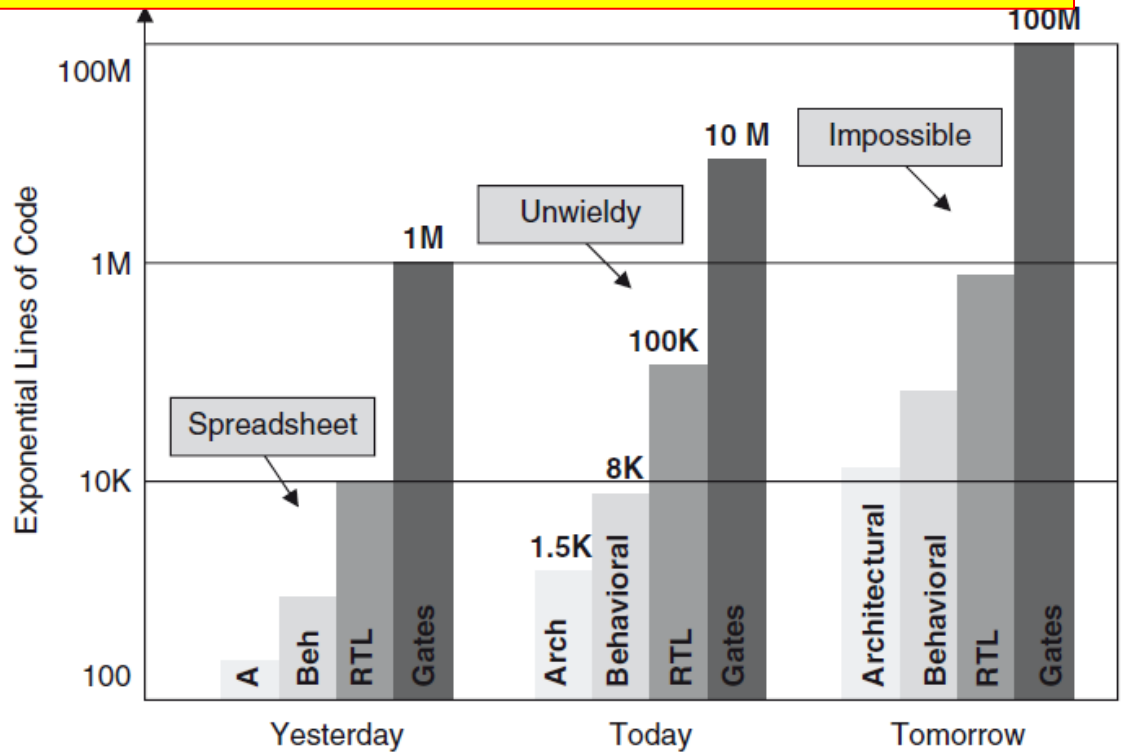




The HW design complexity issues

- Three sample designs from three generations:
  - yesterday, today, and tomorrow.
- The bars for each generation imply the code complexity for four common levels of abstraction associated with system hardware design.

It considers only a single integrated circuit. It does not reflect complexity of a system with several large chips and gigabytes of application software.

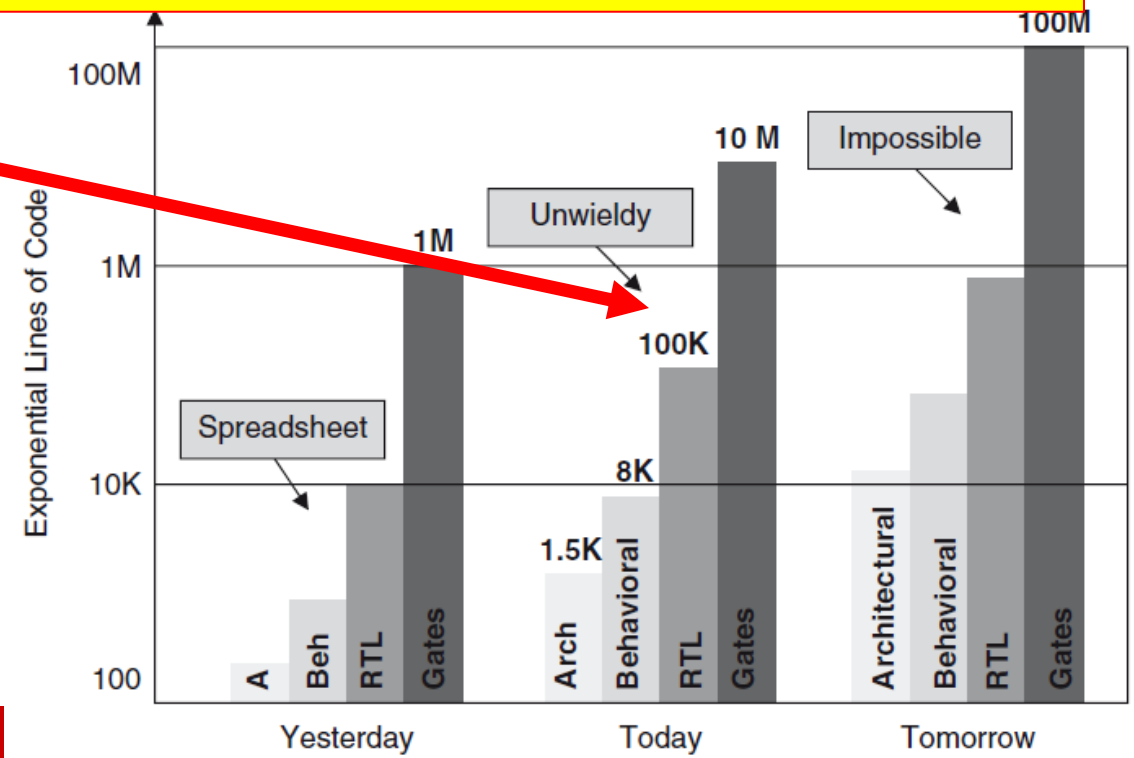


The HW design complexity issues

- Three sample designs from three generations:
  - yesterday, today, and tomorrow.
- The bars for each generation imply the code complexity for four common levels of abstraction associated with system hardware design.

It considers only a single integrated circuit. It does not reflect complexity of a system with several large chips and gigabytes of application software.

Today's designs are practical because of the methodologies that apply RTL synthesis for automated generation of gates.



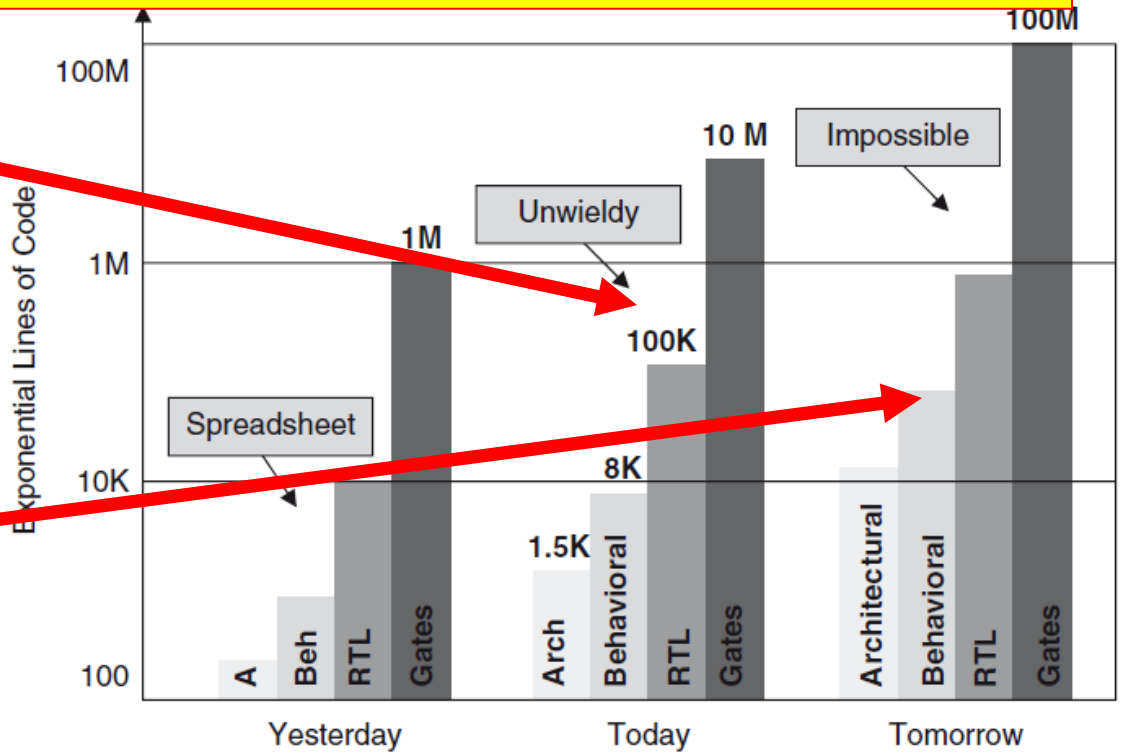
The HW design complexity issues

- Three sample designs from three generations:
  - yesterday, today, and tomorrow.
- The bars for each generation imply the code complexity for four common levels of abstraction associated with system hardware design.

It considers only a single integrated circuit. It does not reflect complexity of a system with several large chips and gigabytes of application software.

Today's designs are practical because of the methodologies that apply RTL synthesis for automated generation of gates.

Tomorrow's integrated circuits will exceed one hundred million gates. This size equates to roughly one million lines of RTL code, if written using today's methodologies.



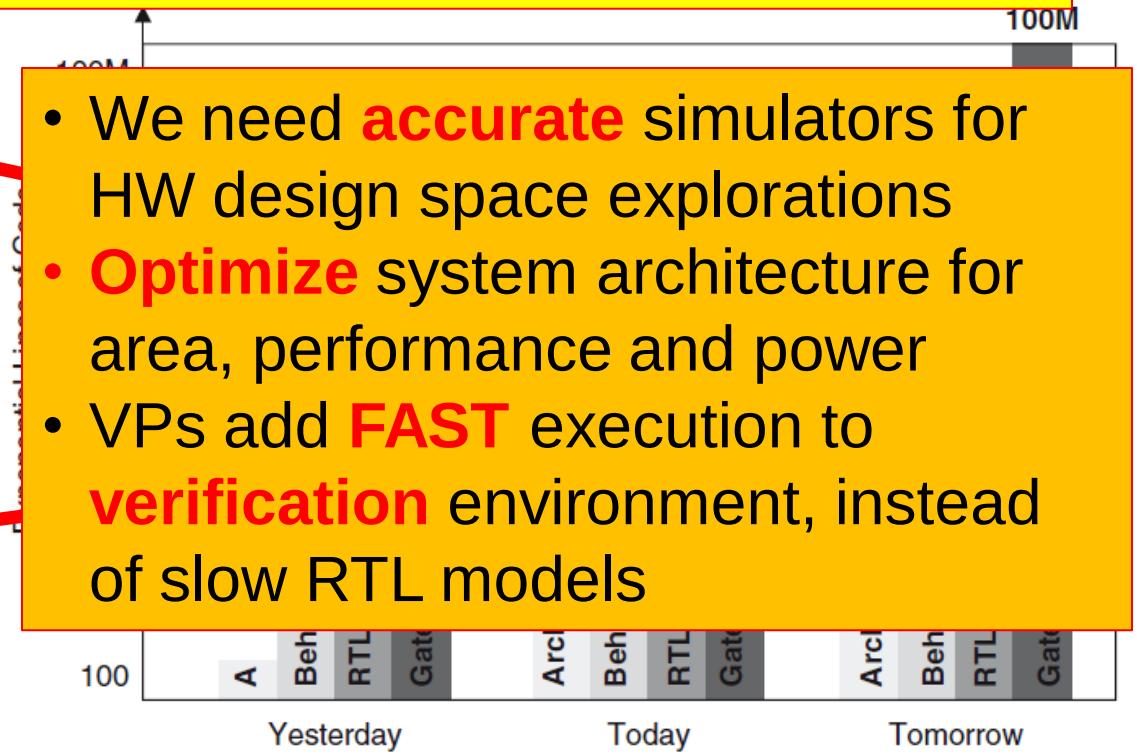
The HW design complexity issues

- Three sample designs from three generations:
  - yesterday, today, and tomorrow.
- The bars for each generation imply the code complexity for four common levels of abstraction associated with system hardware design.

It considers only a single integrated circuit. It does not reflect complexity of a system with several large chips and gigabytes of application software.

Today's designs are practical because of the methodologies that apply RTL synthesis for automated generation of gates.

Tomorrow's integrated circuits will exceed one hundred million gates. This size equates to roughly one million lines of RTL code, if written using today's methodologies.



- We need **accurate** simulators for HW design space explorations
- **Optimize** system architecture for area, performance and power
- VPs add **FAST** execution to **verification** environment, instead of slow RTL models

Virtual Platforms for HW Design Space Explorations

- Optimize system architecture for area, performance and power
  - What is the expected latency and clock speed?
  - What additional control and status is easy and inexpensive to provide to software?
  - What is the estimated power consumption of this block?
  - Is there enough performance in the bus architecture to implement our algorithm in hardware and realize performance gains?
  - Is the bus performance adequate? Will we require multiple busses or a multi-tiered bus concept?
  - Is the arbitration scheme sufficient?
  - Is the estimated size and cost within our product goals?
  - Would a different processor help performance? How would a lower clock rate or less capable processor affect performance?

The SW design complexity issues

Software complexity is increasing exponentially

Software code for embedded systems is doubling annually

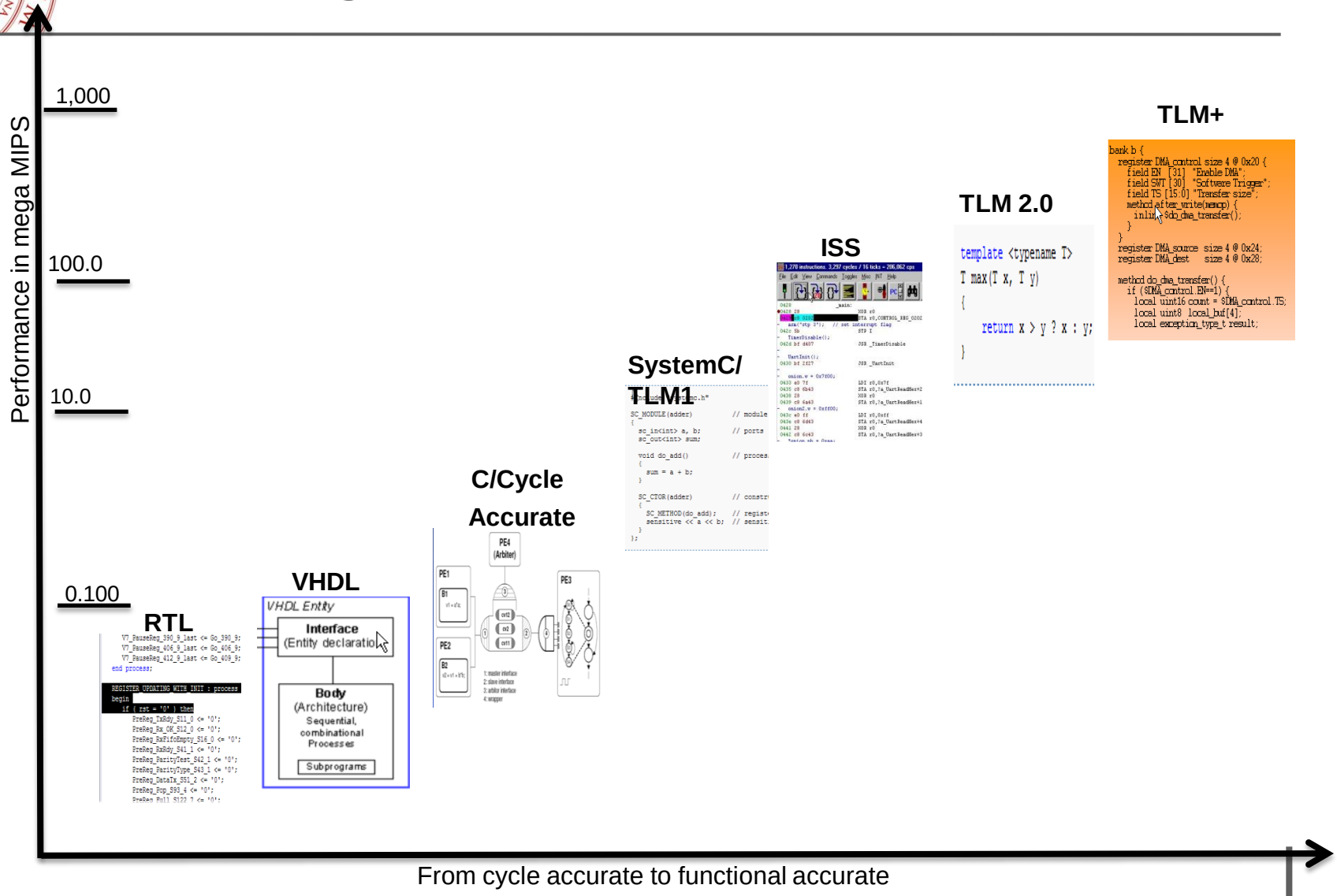
Amount of concurrency is doubling every 18 months

- **Complexity of software driven by concurrency**
  - shared resources: communication arch., memories, I/O, etc.

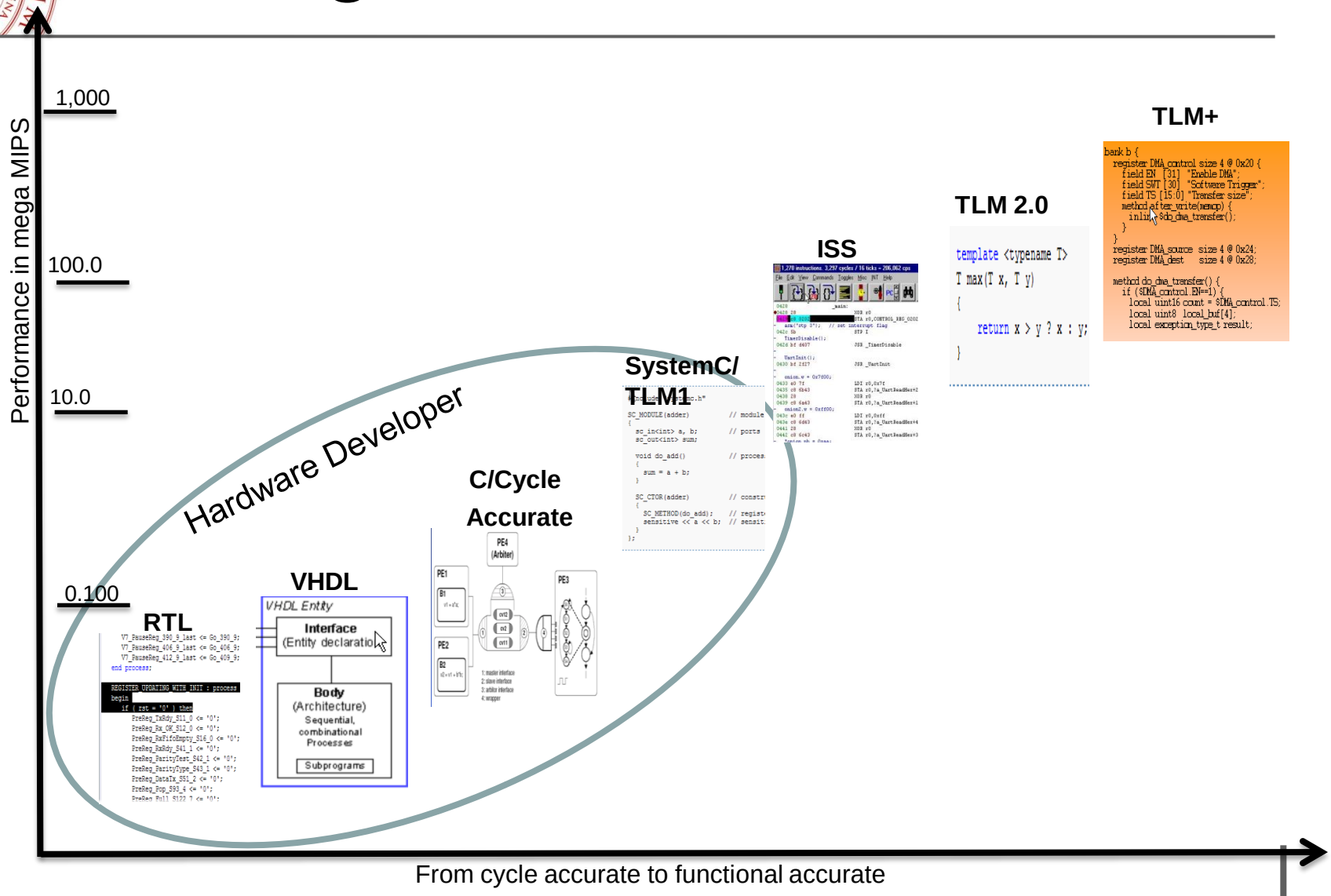
New technologies, methodologies needed for functional and performance verification of embedded software

- If **initial testing of software** is done on Virtual Platforms, could reduce SoC schedule by months
  - **Reducing** initial development and maintenance **costs** significantly
  - Run complete OS and SW stack during system level verification
  - Verifying of SW in a controlled environment
- Moving to Virtual Platforms for earlier testing of software
  - Does the HW have the **advertised features**?
  - Can the SW **meet its timing budget** with the current architecture?
  - Does the SW have **the necessary control and observably** to meet the customer requirements?

The right level of abstraction

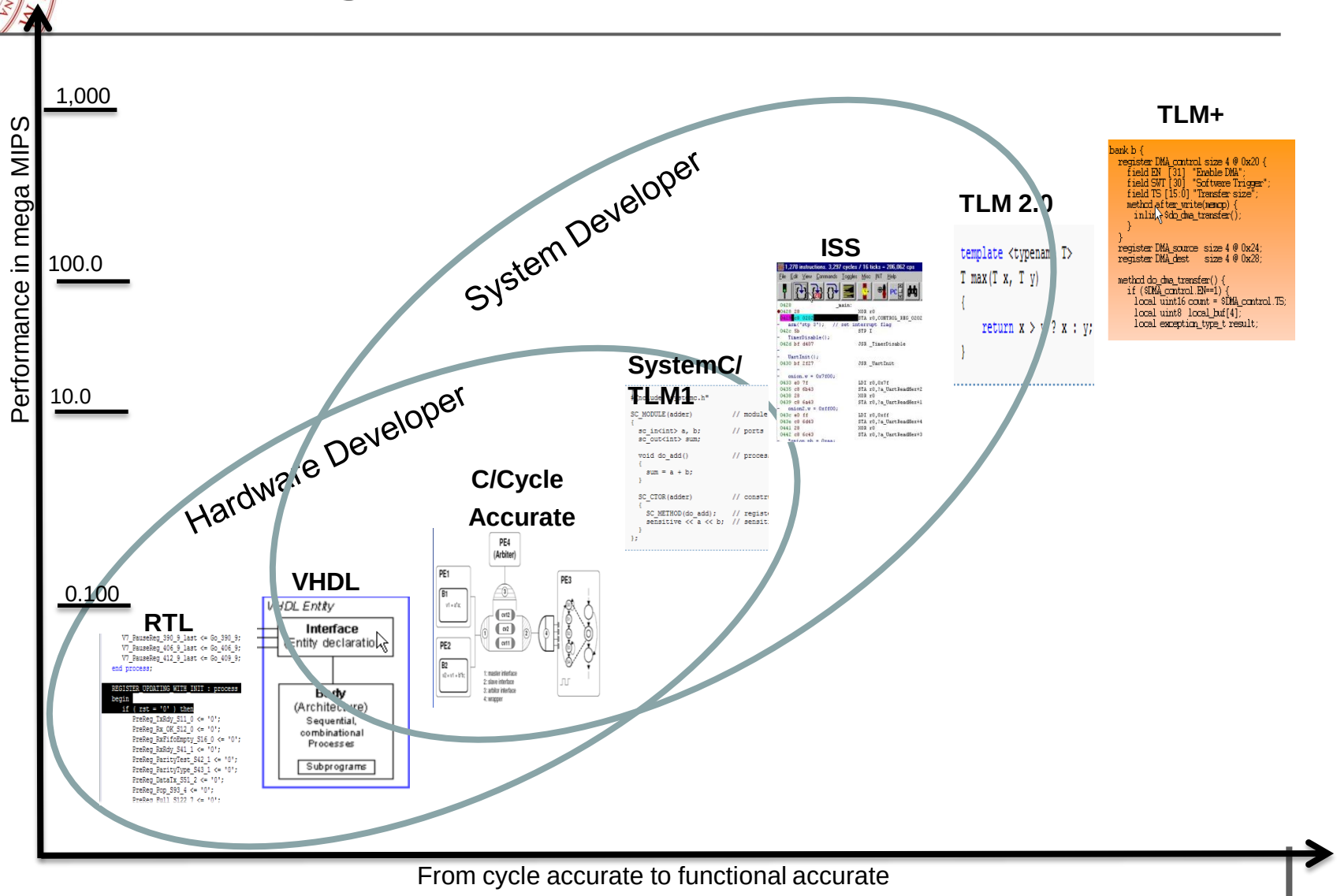


The right level of abstraction

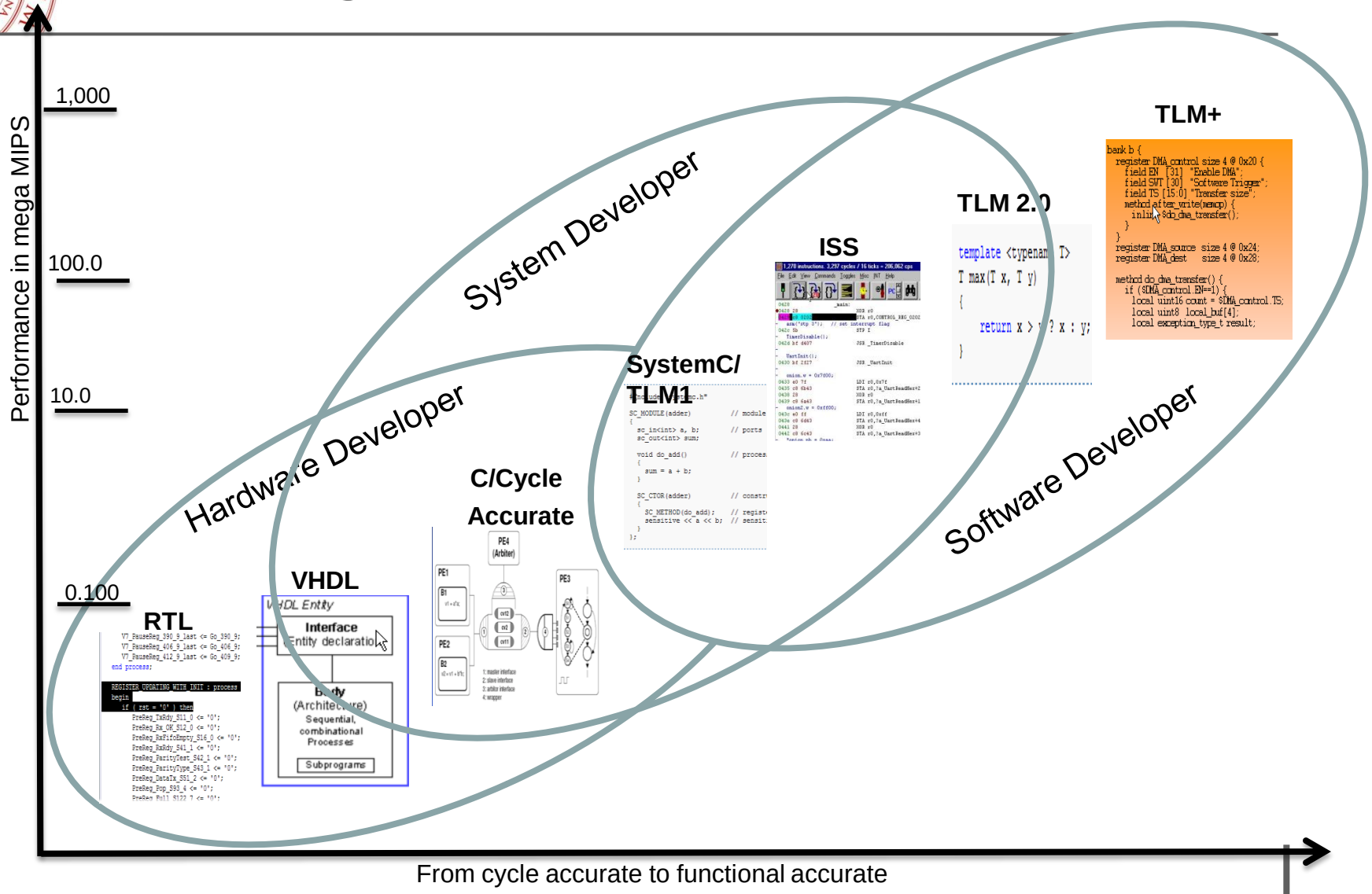


From cycle accurate to functional accurate

The right level of abstraction



The right level of abstraction



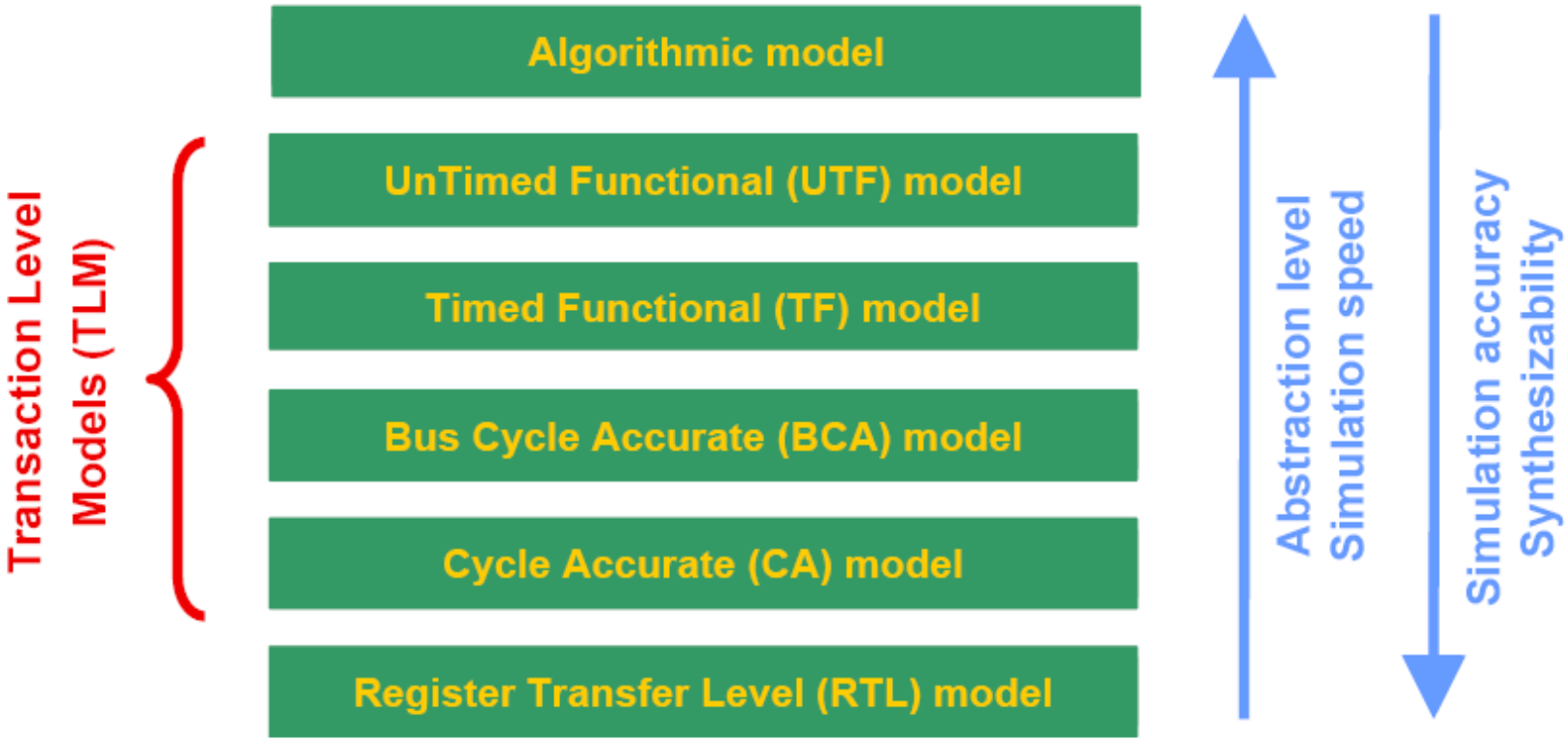
SystemC

Francesco Paterna, Martino Ruggiero, Luca Benini

DEIS – Micrel lab

francesco.paterna@unibo.it

Layers of Hardware Design



Scope of Layers

- Algorithmic model
- UnTimed Functional (UTF) model

No notion of time (processes and data transfers)

- Timed Functional (TF) model
- Bus Cycle Accurate (BCA) model

Notion of time (processes and data transfers)

- Cycle Accurate (CA) model
- Register Transfer Level (RTL) model

Cycle accuracy, signal accuracy

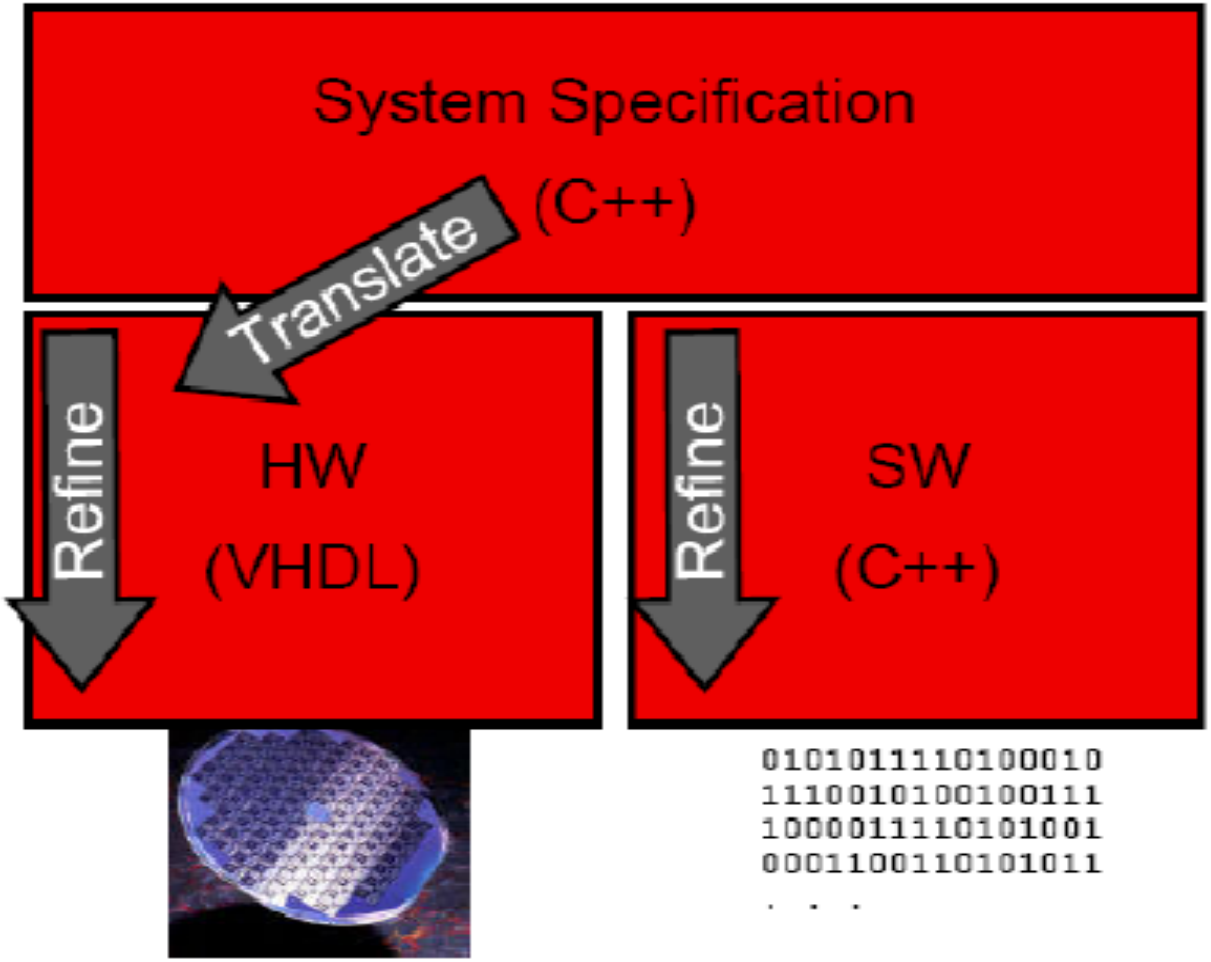
Purpose of Layers

Algorithmic model	✓ Functional verification ✓ Algorithm validation
UnTimed Functional (UTF) model	
Timed Functional (TF) model	✓ Coarse benchmarking ✓ Application SW development ✓ Architectural analysis
Bus Cycle Accurate (BCA) model	
Cycle Accurate (CA) model	✓ Detailed benchmarking ✓ Driver development ✓ Microarchitectural analysis
Register Transfer Level (RTL) model	

What language?

- Problem: very different levels of abstraction
- Modeling language to use?
- High-level: Java, Visual Basic, C++
- Low-level: HDLs (VHDL,Verilog)
- Huge modeling gap
- Need to translate models…

System Design Methodology



Why not just C++?

Concurrency support is missing (HW is inherently parallel)

No notion of time (clock, delays)

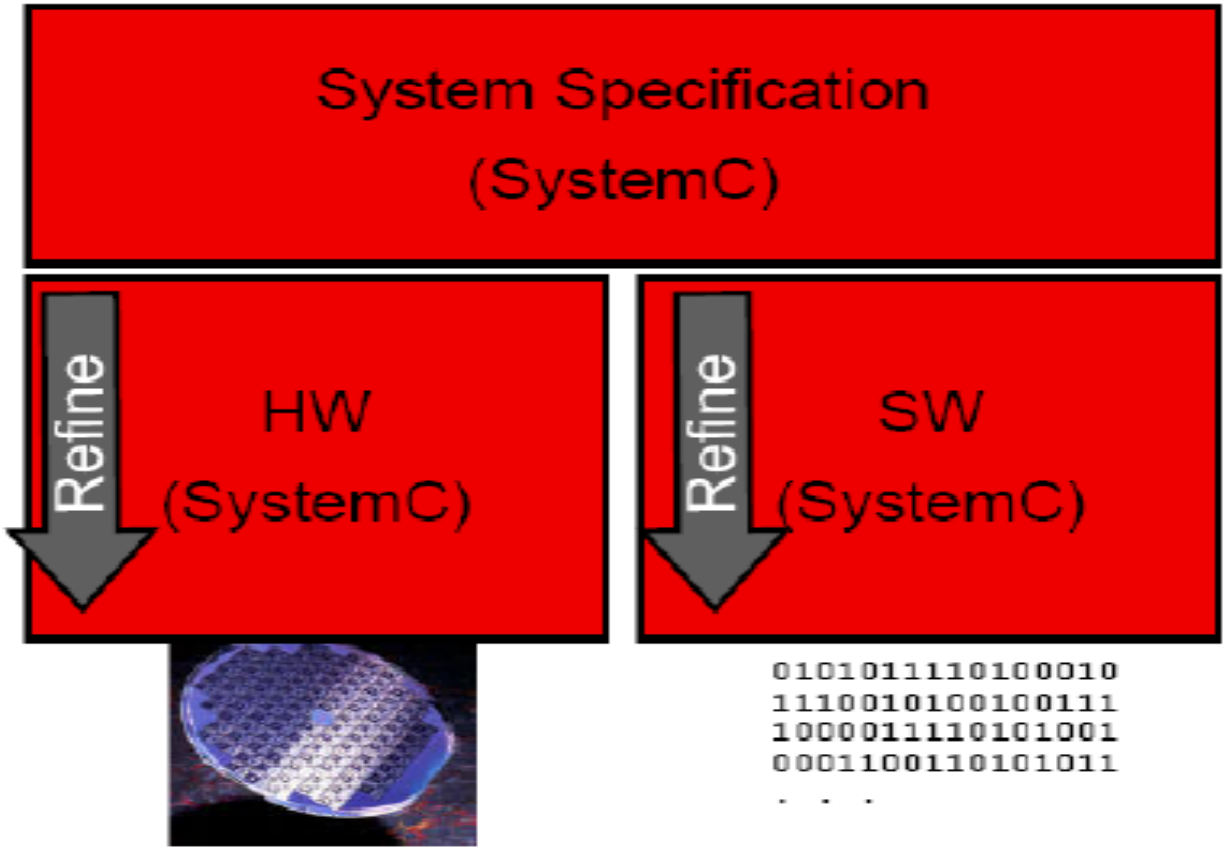
Communication model is very different from actual HW model (signals)

Weak/complex reactivity to events

Missing data types (logic values, bit vectors, fixed point math)

Plain vanilla C++ not viable!

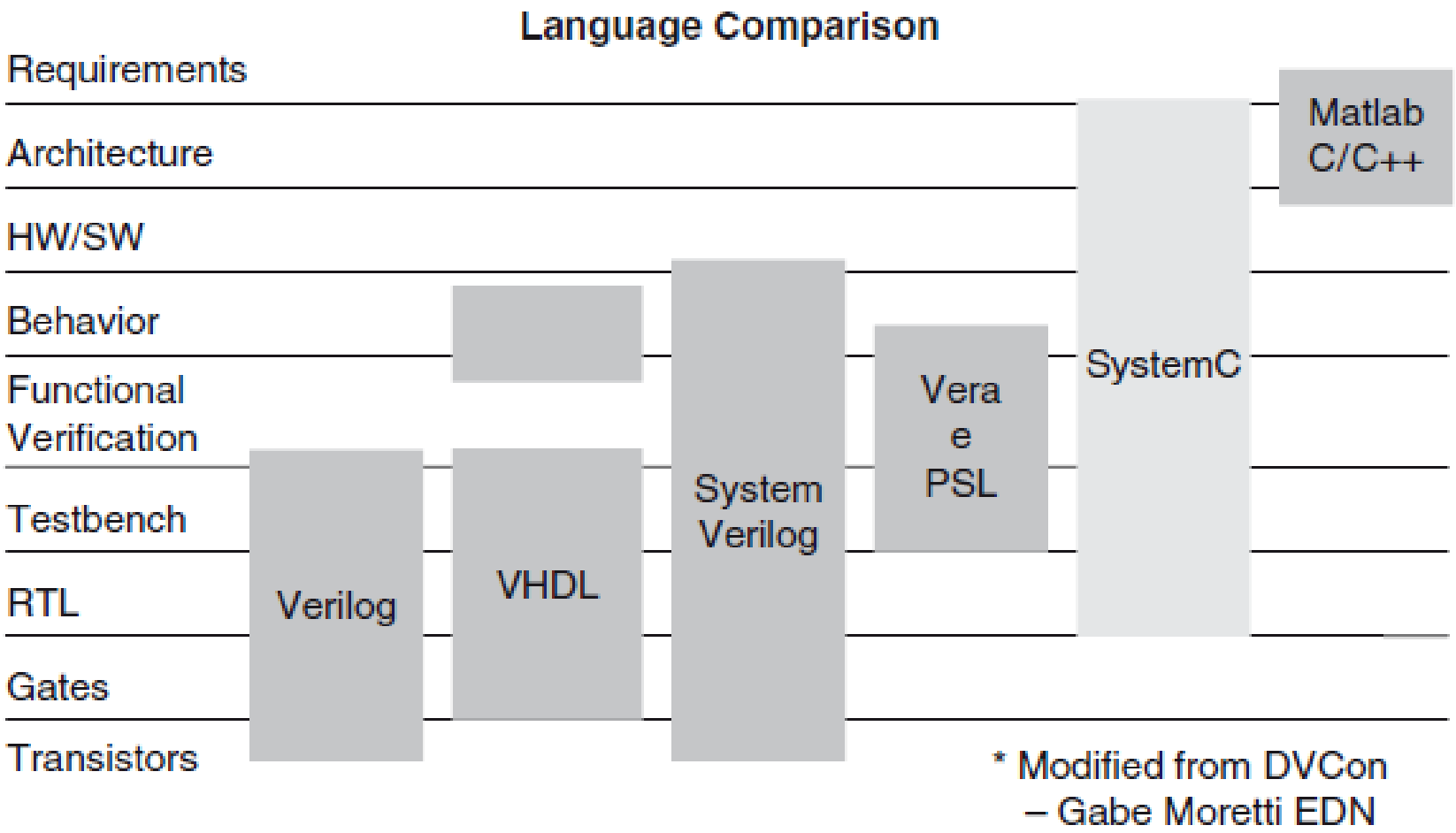
SystemC


System Specification
(SystemC)
Refine
HW
(SystemC)
Refine
SW
(SystemC)
0101011110100010
1110010100100111
1000011110101001
0001100110101011
. . .

SystemC

- Objectives:
  - Model Hw with Sw programming language
  - Achieve fast simulation
  - Provide support for hw/sw system design
- Requirement:
  - Give hw semantics to sw models
- Supported by a large consortium of semiconductor and EDA companies

Language Comparisons and Levels of Abstraction



Introduction to C++
Object Oriented Programming

a complete guide in Italian language is online at:
(http://www.bo.cnr.it/corsi-di-informatica/corsoCstandard/Lezioni/01Indice.html)

Classes in C++

- A ***Class*** is a datatype containing *variables, constants, functions*
- An ***Object*** is an istance of a Class

```
class  CA {

   int a;

   int get_a();
   int set_a(int a_temp);
}

int CA::get_a(){ return a;}
int CA::set_a(int a_temp){ a = a_temp;}
```

```
int main() {
   CA A;
   A.set_a(3);
   int b = A.get_a();
   A.a = 4;
   int c = A.a;
}
```

Classes in C++

- A ***Class*** is a datatype containing *variables, constants, functions*
- An ***Object*** is an istance of a Class

```
class  CA {
  private:
    int a;
   public:
    int get_a();
    int set_a(int a_temp);
}

int CA::get_a(){ return a;}
int CA::set_a(int a_temp){ a = a_temp;}
```

```
int main() {
    CA A;
    A.set_a(3);
    int b = A.get_a();
    A.a = 4;
    int c = A.a;
}
```

Classes in C++

- A ***Class*** is a datatype containing *variables, constants, functions*
- An ***Object*** is an istance of a Class

```
class  CA {
  private:
    int a;
   public:
    int get_a();
    int set_a(int a_temp);
}

int CA::get_a(){ return a;}
int CA::set_a(int a_temp){ a = a_temp;}
```

```
int main() {
    CA A;
    A.set_a(3);
    int b = A.get_a();
    A.a = 4;
    int c = A.a;
}
```

Classes vs. Structures:

1. Member-functions
2. Data hiding

Constructors and Destructors

- Particular member-functions to *initialize* and *release* resources respectevly
- No return type specified
- At most 1 destructor

```
class  CA {
public:
  int* a;
  CA();
  ~CA();

}
```

```
CA::CA(){ a = new int}
CA::~CA(){delete  a;}
```

```
int main() {
    CA A;
    A->a = 5;
   return 0;
}
```

Constructors and Destructors

- Particular member-functions to *initialize* and *release* resources respectevly
- No return type specified
- At most 1 destructor

```
class  CA {
public:
  int* a;
  CA();
  ~CA();
  CA(int value);
}
```

```
CA::CA(){ a = new int}
CA::~CA(){delete  a;}
CA::CA(int value)
{ a = new int;
  *a = value}
```

```
int main() {
    CA A;
    A->a = 5;
   return 0;
}
```

```
int main() {
    CA A(5);
    return 0;
}
```

Inheritance in C++

- Inheritance allows to create classes which are derived from other classes
- A **derived** class includes their **parents**' members to their own

```
class  CA {
private:
int a;
public:
int get_a();
}
```

```
class CB:CA {
private:
 int b;
public:
int get_b();
}
```

```
int main() {
   CB B;
   int c = B.get_a();
   int d = B.get_b();
  return 0;
}
```

Abstract classes in C++

```
class  CA {
public:
virtual int get() = 0;
}
```

Abstract class:
~~CA A;~~

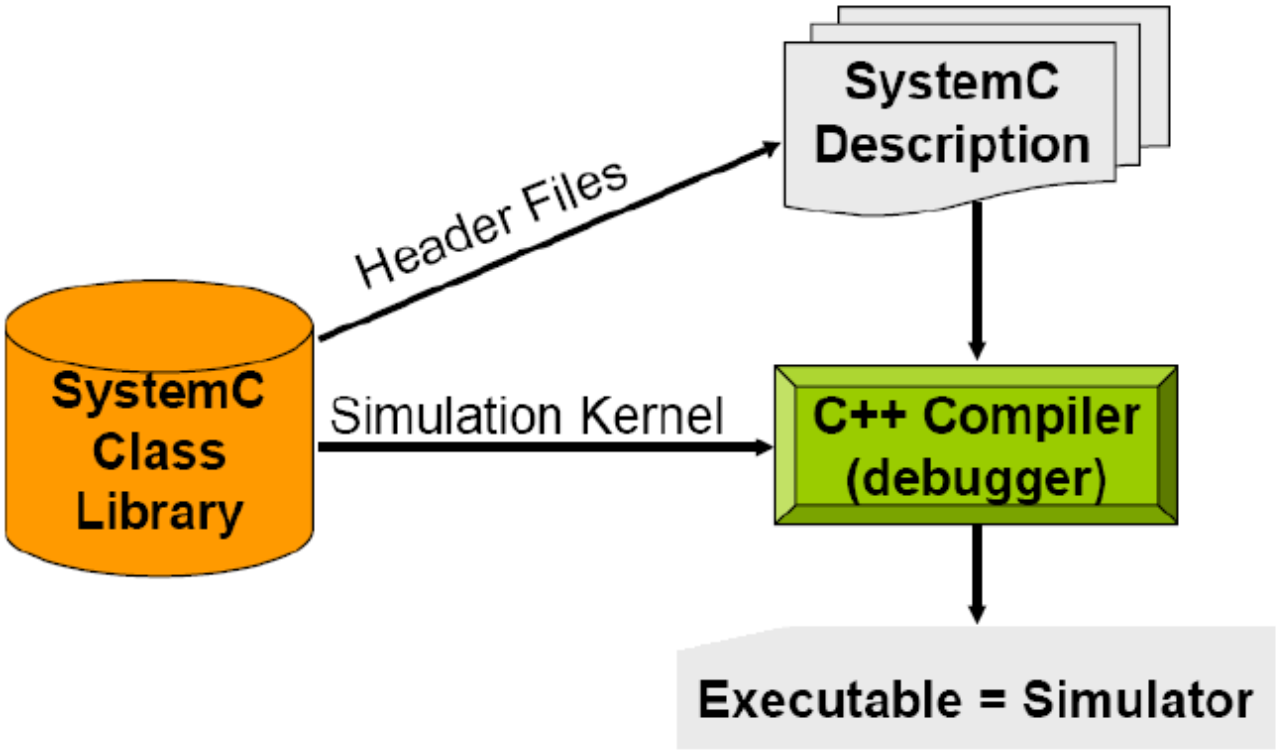
```
class CB:CA {
private:
 int b;
public:
virtual int get() {return b};
}
```

```
class CC:CA {
private:
 int c;
public:
virtual int get() {return c};
}
```

SystemC

- C++ extensions
  - C++ class library (libsystemc.a) and modeling methodology
  - Hw semantics defined through the class library
  - SystemC code is C++ code and can be freely mixed with C++ code
- Object-oriented style
  - Basic component models
  - Inheritance or encapsulation
- Some hw synthesis support
  - Usually translation into traditional HD languages

SystemC infrastructure



SystemC features

- SystemC is able to model HW platforms without extending C++ language syntax

Concurrency	**Processes**
Hardware Data Types	**bit vectors, arbitrary precision signed and unsigned integers, fixed-point numbers**
Notion of Time	**Clocks, wait calls**
Reactive Behavior	**Watching, sensitivity list, events**
Communication	**Signals, protocols**

SystemC Core Language

High-Level Channels

Kahn Process Networks, Master/Slave libraries, …

Elementary channels

Signals, Timers, Mutexes, Semaphores, FIFOs, …

Core language

Modules/Processes
Ports/Interfaces
Events
Channels
Event-driven simulation kernel

Data types

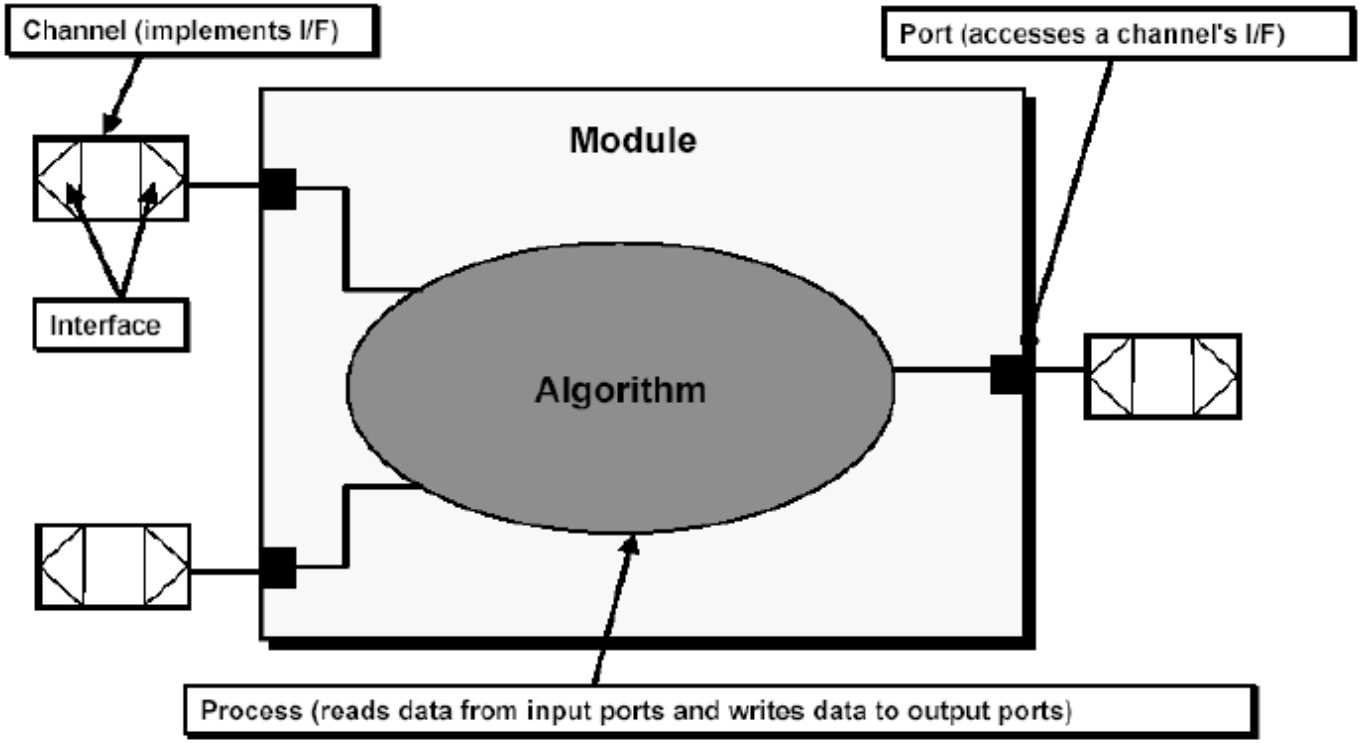
4-valued logic types (01XZ)
Bit/logic vectors
Arbitrary precision integers
Fixed point
C++ user-defined

C++

SystemC Time Model

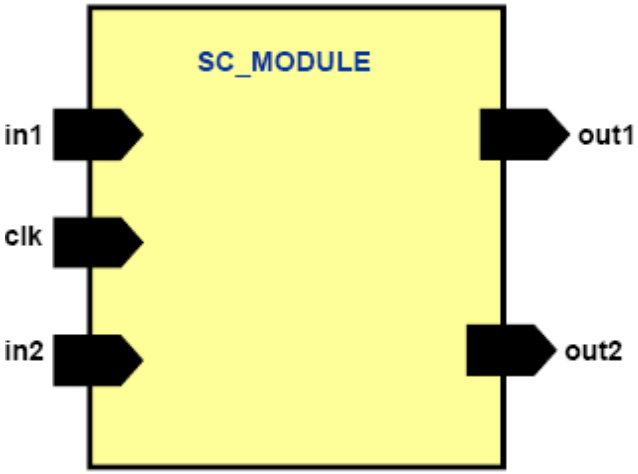
- SystemC uses an integer-valued time model;
- The time is represented by a 64 bit unsigned integer;
- Time has a resolution;
- The minimal time unit is 1 picosecond;
- The time resolution can be set by calling the sc_set_time_resolution() function
  - Ex: sc_set_time_resolution(10,SC_PS);
- sc_time is used to define time durations
  - sc_time t1(42, SC_NS);

SystemC classes



SystemC Classes: Modules

- Modules (sc_module)
  - Fundamental structural entity
  - Map functionality of HW/SW blocks
  - Contain processes (functionality)
  - Contain other modules (creating hierarchy)
  - Communicate through ports/interfaces/signals

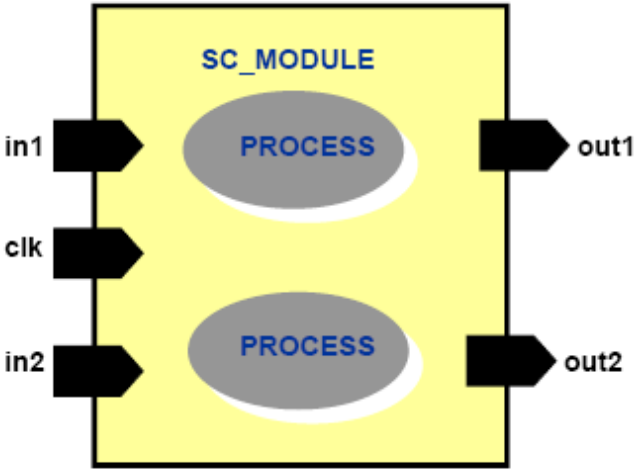


Modules

```
//my_module.h

SC_MODULE(my_module)
{
        //port declarations
        //process declarations

        SC_CTOR(my_module)
        {
                //process configuration
                //initialization code
        }
};
```

SystemC Classes - Processes

- Processes
  - Functionality is described in a process
  - Implemented as member functions
  - Processes run concurrently
  - Code inside a process executes sequentially
  - SystemC has three different types of processes
    - SC_METHOD
    - SC_THREAD
    - SC_CTHREAD
  - Macros telling the scheduler how to handle such processes



Process types

- **sc_method: method process**
  - sensitive to a set of signals
  - Cannot be suspended: executed until it returns
  - Does not remember internal state across invocations (unless explicitly kept in member variables)
- **sc_thread: thread process**
  - sensitive to a set of signals
  - Can be suspended: executed until a wait(), resumed through sensitivity list
  - Remembers internal state across invocations
- **sc_cthread: clocked thread process**
  - sensitive only to one edge of clock
  - execute until a wait() or a wait_until() (note the enhanced wait semantics)
  - watching(reset) restarts from top of process body (reset evaluated on active edge)
  - Used mainly in high level simulation

SC_METHOD

```
//my_module.h

SC_MODULE(my_module)
{
  sc_in<bool> id;
  sc_in<sc_uint<3> > in_a;
  sc_in<sc_uint<3> > in_b;
  sc_out<sc_uint<3> > out_c;
  void my_method();
  SC_CTOR(my_module)
  {
    SC_METHOD(my_method);
    sensitive << a << b;
  }
};
```

```
//my_module.cpp

void my_module::my_method()
{
  if (id.read())
    out_c.write(in_a.read());
  else
    out_c.write(in_b.read());
};
```

SC_THREAD

```
//my_module.h

SC_MODULE(my_module)
{
  sc_in<bool> id;
  sc_in<bool> clock;
  sc_in<sc_uint<3> > in_a;
  sc_in<sc_uint<3> > in_b;
  sc_out<sc_uint<3> > out_c;
  void my_thread();
  SC_CTOR(my_module)
  {
    SC_THREAD(my_thread);
    sensitive << clock.pos();
  }
};
```

```
//my_module.cpp

void my_module:: my_thread()
{
  while(true)
  {
    if (id.read())
      out_c.write(in_a.read());
    else
      out_c.write(in_b.read());
    wait();
  }
};
```

wait_until()

```
do
      wait();
while (a.read() != true || b.read() != true);
```

```
wait_until(a.delayed() == true && b.delayed() == true);
```

watching()

```
//my_module.cpp

void my_module::my_cthread()
{
    while(true)
    {
        […]
        if (reset.read())
         [reset code]
        […]
        if (reset.read())
          [reset code]
        […]
        wait();
    }
}
```

```
//my_module.h
SC_CTOR(my_module)
{
  SC_CTHREAD(my_cthread, clock.pos());
  watching(reset.delayed() == true);
}
```

```
//my_module.cpp
void my_module::my_cthread()
{
    [reset code]
    while(true)
    {
            […]
            wait();
    }
}
```

watching(): example

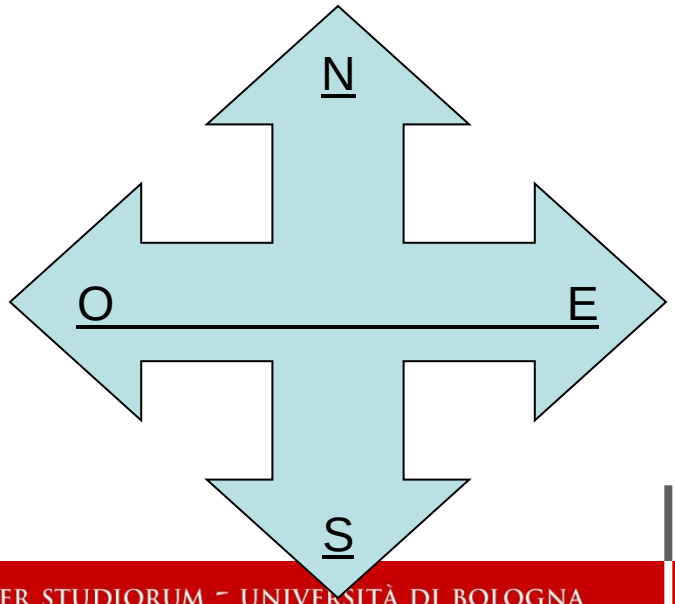
```
#include "systemc.h"

SC_MODULE(data_gen) {
sc_in_clk clk;
sc_inout<int> data;
sc_in<bool> reset;
void gen_data();
SC_CTOR(data_gen){
SC_CTHREAD(gen_data, clk.pos());
watching(reset.delayed() == true);
}
};
```

```
#include "datagen.h"

void gen_data() {
if (reset == true) {
data = 0;
}
while (true) {
data = data + 1;
wait();
data = data + 2;
wait();
data = data + 4;
wait();
}
}
```

Example: semaphore (1/3)

```
// traff.h
#include "systemc.h"
SC_MODULE(traff) {

sc_in<bool> roadsensor;
sc_in<bool> clock;
sc_out<bool> NSred;
sc_out<bool> NSyellow;
sc_out<bool> NSgreen;
sc_out<bool> EWred;
sc_out<bool> EWyellow;
sc_out<bool> EWgreen;

void control_lights();
int i;
```

```
// Constructor
SC_CTOR(traff) {
 SC_THREAD(control_lights);
  // Thread Process
  sensitive << roadsensor;
  sensitive_pos << clock;
 }
};
```



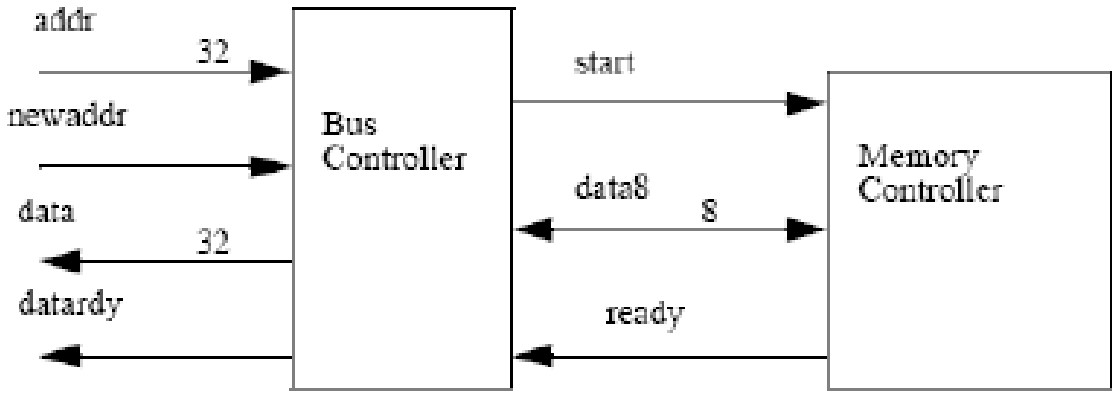
Example: semaphore (2/3)

```
#include "traff.h"
void traff::control_lights() {
NSred = false;
NSyellow = false;
NSgreen = true;
EWred = true;
EWyellow = false;
EWgreen = false;
while (true) {
  while (roadsensor == false)
     wait();
  NSgreen = false; //road sensor
  NSyellow = true; //set NS to yellow
  NSred = false;
  for (i=0; i<5; i++)
     wait();
NSgreen = false;
NSyellow = false;
NSred = true;
EWgreen = true;
EWyellow = false;
EWred = false;
for (i= 0; i<50; i++)
  wait();
```

Example: semaphore (3/3)

```
    NSgreen = false; // times up for EW green
    NSyellow = false; // set EW to yellow
    NSred = true;
    EWgreen = false;
    EWyellow = true;
    EWred = false;
    for (i=0; i<5; i++) // times up for EW yellow
        wait();
    NSgreen = true; // set EW to red
    NSyellow = false; // set NS to green
    NSred = false;
    EWgreen = false;
    EWyellow = false;
    EWred = true;
    for (i=0; i<50; i++) // wait one more long
        wait();          // interval before allowing
                         // a sensor again
  }
}
```

SC_CTHREAD: example



SC_CTHREAD: example (1/3)

```
#include "systemc.h"
SC_MODULE(bus) {
   sc_in_clk clock;
   sc_in<bool> newaddr;
   sc_in<sc_uint<32> > addr;
   sc_in<bool> ready;
   sc_out<sc_uint<32> > data;
   sc_out<bool> start;
   sc_out<bool> datardy;
   sc_inout<sc_uint<8> > data8;

   sc_uint<32> tdata;
   sc_uint<32> taddr;

   void xfer();

   SC_CTOR(bus) {
    SC_CTHREAD(xfer, clock.pos());
    datardy.initialize(true);}
};
```

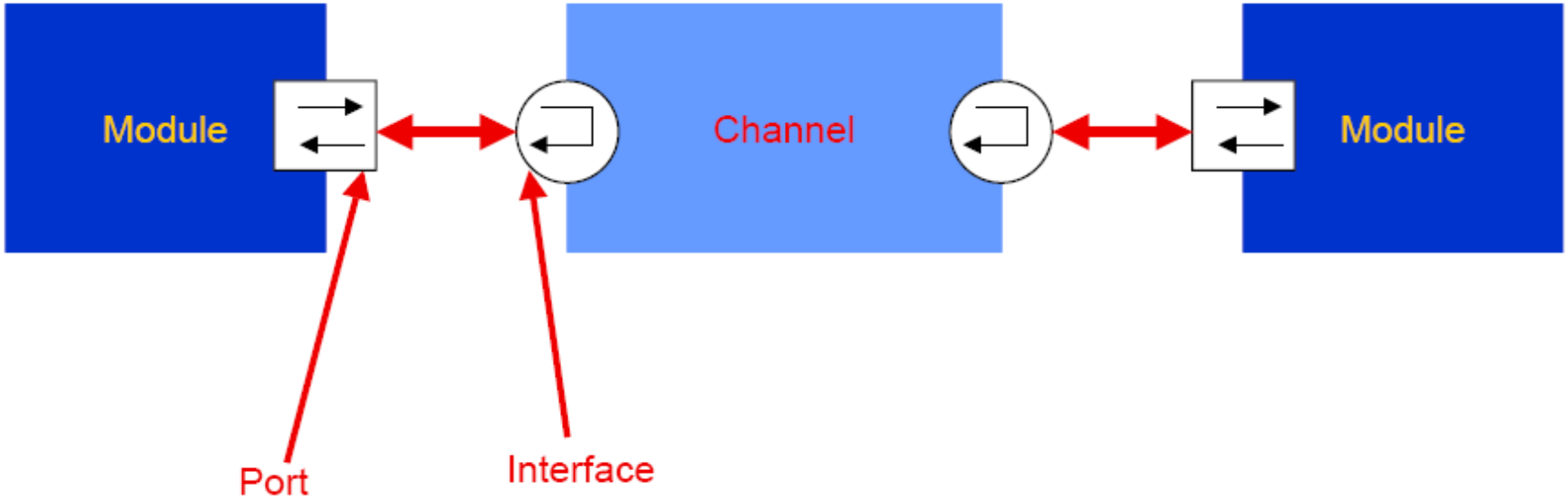
SC_CTHREAD: example (2/3)

```
#include "bus.h"
void bus::xfer() {
   while (true) {
   // wait for a new address to appear
   wait_until( newaddr.delayed() == true);
   // got a new address so process it
   taddr = addr.read();
   datardy.write(false); // cannot accept new address now
   data8 = taddr.range(7,0);
   start.write(true); // new addr for memory controller
   wait();
   // wait 1 clock between data transfers
   data8 = taddr.range(15,8);
   start.write(false);
   wait();
```

SC_CTHREAD: example (3/3)

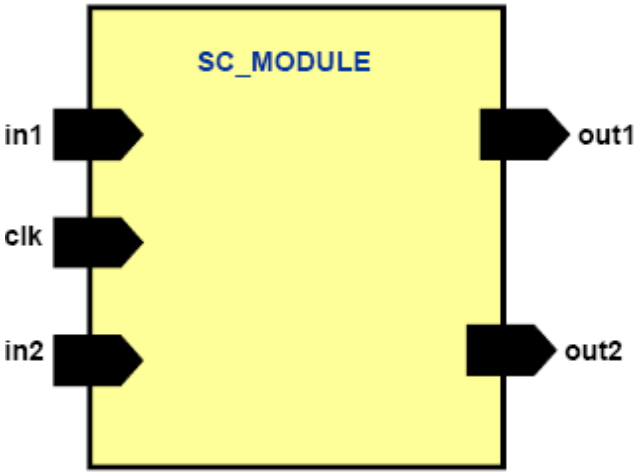
```
  data8 = taddr.range(23,16);
  wait();
  data8 = taddr.range(31,24);
  wait();
  // now wait for ready signal from memory controller
  wait_until(ready.delayed() == true);
  // now transfer memory data to databus
  tdata.range(7,0) = data8.read();
  wait();
  tdata.range(15,8) = data8.read();
  wait();
  tdata.range(23,16) = data8.read();
  wait();
  tdata.range(31,24) = data8.read();
  wait();
  data = tdata;
  datardy = true; // data is ready, new addresses ok
 }
}
```

Ports, Interfaces, Channels



SystemC Classes: Ports

- Ports (sc_in<>,sc_out<>,sc_inout<>)
  - Modules have ports
  - Ports have types
  - A process can be made sensitive to ports/signals



Ports

- They provide communication functions to modules
- Derived from SystemC class sc_port<class IF,intN=1> (type of interface, number of connected interfaces)
- On the outside, they connect to channels by means of interfaces
- Typical channel (in RTL models): sc_signal
- In this case, shortcuts exist: sc_in<class T>, sc_out<class T>,sc_inout<class T>

  (ports connected to N=1interfaces of type sc_signal_in_if<class T>)
- Methods made available by the underlying interface:
  - my_port.read(), my_port.write(), …

Ports

```
//my_module.h

SC_MODULE(my_module)
{
        sc_in<bool> id;
        sc_in<sc_uint<3> > in_a;
        sc_in<sc_uint<3> > in_b;
        sc_out<sc_uint<3> > out_c;
        //process declarations
        SC_CTOR(my_module)
        {
                //process configuration
                //initialization code
        }
};
```

Communication semantics

- **Interface Method Calls (IMC)**
  - Process calls an interface method of a channel
  - The collection of a fixed set of communication **Methods** is called an **Interface** (virtual objects)
  - More than one channel is feasible for the same **Interface**
  - **Modules** can be connected via their **Ports** to those **Channels**

Implementing an Interface

- An interface does not actually do anything
- It just has to declare prototypes for the channel methods

```
class my_read_if : virtual public sc_interface
{
public:
   virtual char read() = 0;
};

class my_write_if : virtual public
sc_interface
{
public:
   virtual void write(char) = 0;
};
```

Written this way, a module will only be able to connect to one of the interfaces – either for read or for write!

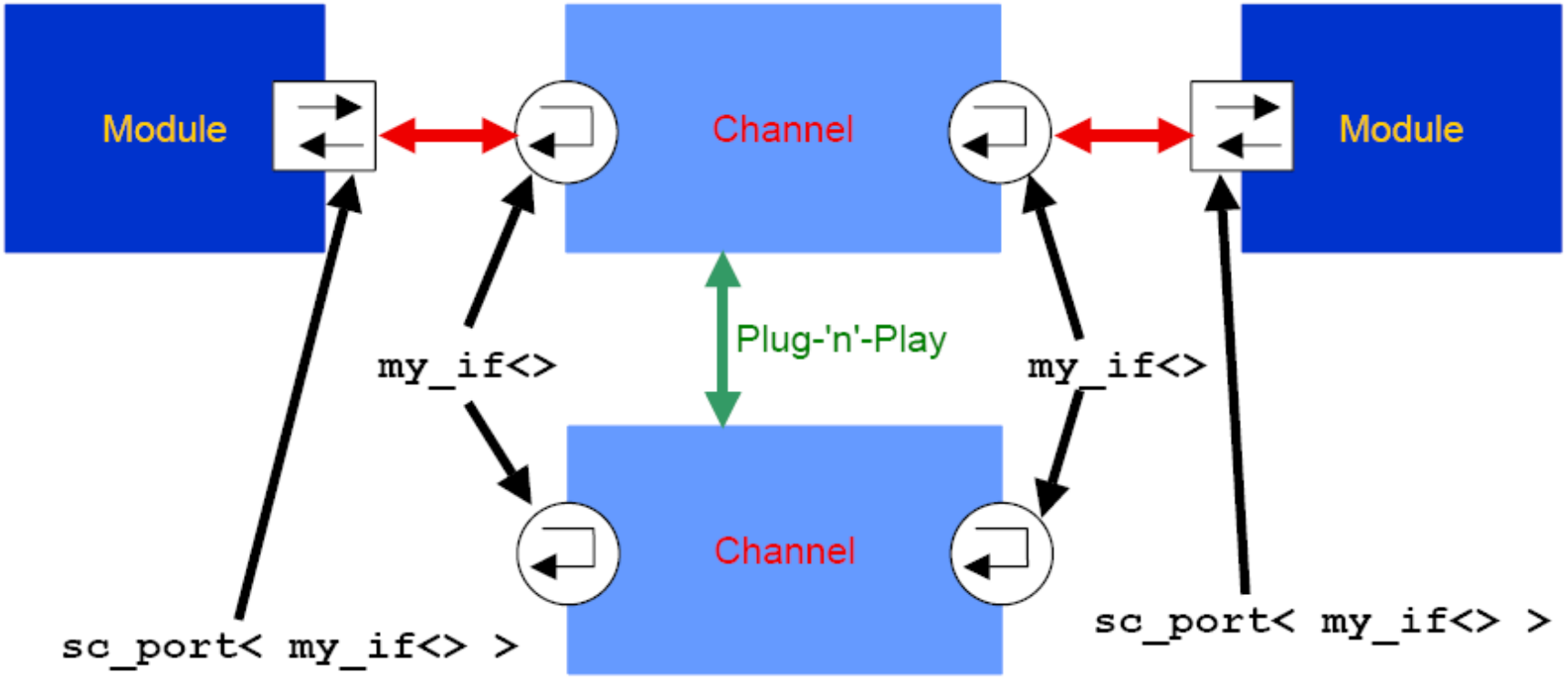
Implementing an Interface

- To be able to implement a read/write connection to the channel, we need a derived interface:

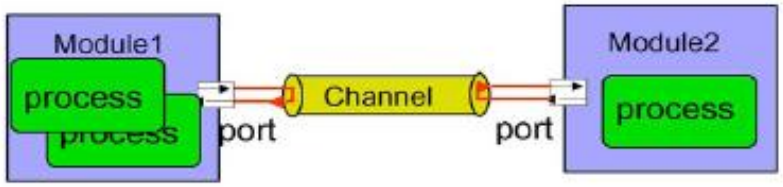
```
class my_readwrite_if :
      public my_read_if,
      public my_write_if
{
};
```

- Now both `read()` and `write()` will be available to a module port connected to `my_readwrite_if`

Channel Binding



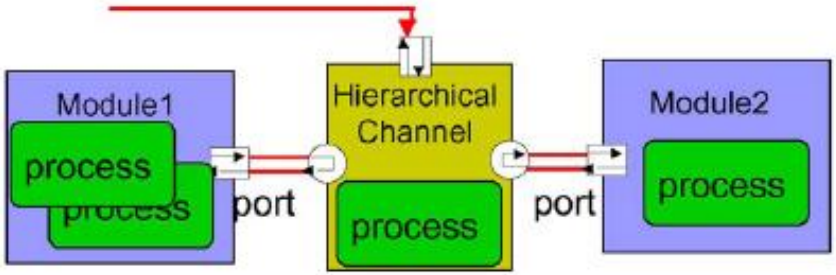
Channels




```
Module1::process()
{
 port->write(42);
}
```

```
Module2::process()
{
 x = port->read();
}
```

Primitive



Hierarchical

- Channels can be
  - primitive (supporting "Evaluate then update" semantics)
  - hierarchical (may include modules, processes, ports, subchannels)
- Code to be implemented:
  - The channel itself
  - The interface

Implementing a Channel

- The channel must provide the implementation of the methods declared in its interfaces
- The channel inherits sc_prim_channel and thus can access its API, especially:
  - void request_update()
  - virtual void update() = 0 (requires an implementation inside of the channel!)
- The channel also inherits sc_interface (through its interfaces) and thus can access its API, especially:
  - virtual const sc_event& default_event() (tells the simulation kernel which is the event on which to enqueue processes waiting on their sensitivity list)

Request-Update Semantics

- The channel must comply with the "Evaluate then update" paradigm of SystemC! To do so:
  - When some channel method is invoked (typically writes), the channel calls the simulation kernel via `sc_prim_channel`'s `request_update()//for VHDL-like simulation`
  - The simulation kernel schedules the channel for update in the next Delta cycle. When such time arrives, the kernel calls the channel's `update()` method
  - `update()` is still inherited by `sc_prim_channel`, but its implementation is channel-specific

Implementing a Channel

```
class my_channel :
   public sc_prim_channel, public my_read_if, public my_write_if
{
   public:
      […]
      virtual char read() { return curr_val };
      virtual void write(char val) { next_val = val;
         if (next_val != curr_val) request_update(); }
      virtual const sc_event& default_event() const
         { return val_chg_event; } //called by S.kern  At the beginn.
   ……
      virtual void update() { curr_val = next_val;
         val_chg_event.notify(SC_ZERO_TIME); }
      char curr_val, next_val;
      sc_event val_chg_event;
};
```

"Evaluate then update" semantics

Sensitivity support

One Delta cycle

Instantiating a Channel

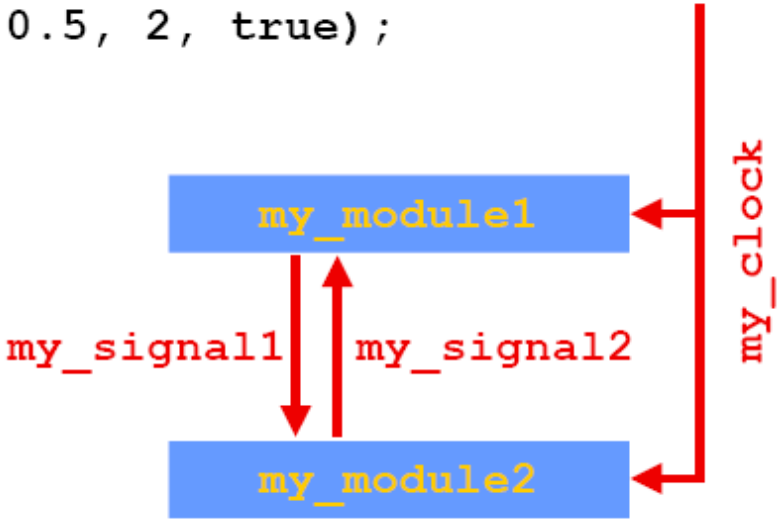
```
// my_module.h

SC_MODULE (my_module)
{
   sc_port <my_read_if> in;
   void my_function();

   SC_CTOR(my_module)
   {
       SC_METHOD(my_function);
       sensitive << in;
   }
};
```

```
// my_module.cpp

void my_module::my_function()
{
   printf("%c\n", in.read());
};
```

```
//main.cpp
My_channel CH;
My_module MYMOD;
MYMOD.in(CH);
```

Signal Examples

```
sc_signal<sc_uint<8> > my_signal1;
sc_signal<bool> my_signal2;
sc_clock my_clock("my_clock", 20, 0.5, 2, true);

my_module_1.clock_port(my_clock);
my_module_1.out_port(my_signal1);
my_module_1.in_port(my_signal2);

my_module_2.clock_port(my_clock);
my_module_2.out_port(my_signal2);
my_module_2.in_port(my_signal1);
```



Read and write from/to port and signal

- It is possible to read and write using either the read() and write() methods or by means of assignment.
- The use of read() and write() is recommended for ports and signals; assignment for internal variables.

```
// read method
 address = into.read();
// assignment
 temp1 = address;
 data_tmp = memory[address];
// write method
 outof.write(data_tmp);
// assignment
 temp2 = data_tmp;
```

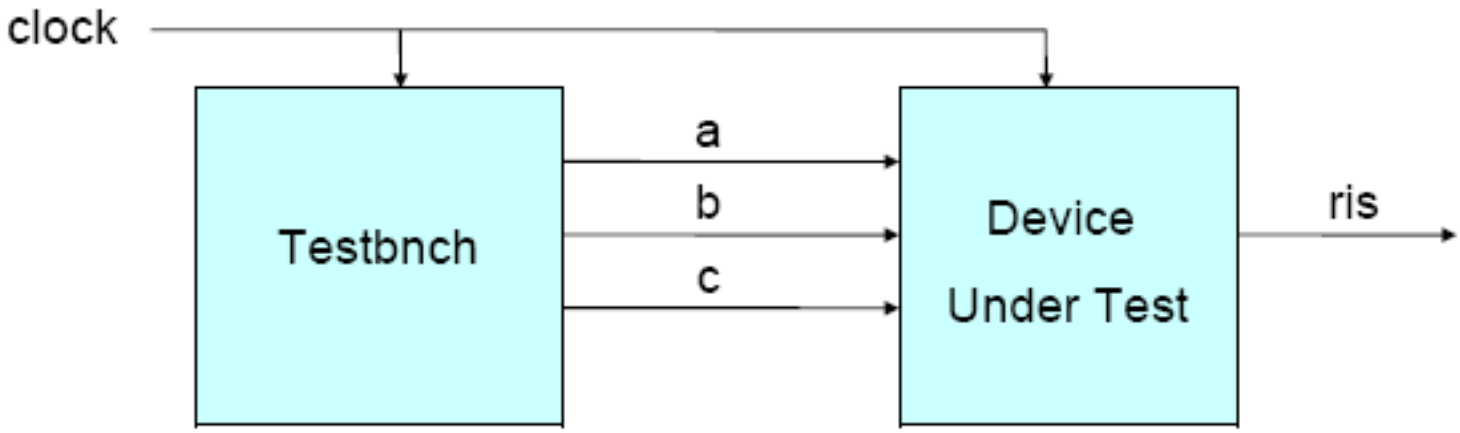
Read and write of single bits of port and signal

- Read and write methods operate on every bit of ports and signals.
- It is not possible to apply read and write methods selectively on single bits.
- To do that it is necessary to use temporary variables.

```
Esempio
//...
sc_signal <sc_int<8> > a;
sc_int<8> b;
bool c;
b = a.read();
c = b[0];
//c=a[0]; Non è consentita in SystemC
```

Simulation

- In order to have a working simulation of a system, it is necessary to define a testbench module.
- Typically the testbench has a CTHREAD process which handles the inputs of the Device Under Test (DUT)



Testbench: example

```
/* tbench.h */

#include "systemc.h"
SC_MODULE(tbench){
 sc_in_clk clock;
 sc_out<int> a;
 sc_out<int> b;
 sc_out<int> c;

 void genera ();

 SC_CTOR(tbench) {
  SC_CTHREAD(genera, clock.pos());
 }
};
```

```
/* tbench.cpp */

 void tbench::genera()
{
  a.write(5);
  b.write(3);
  c.write(2);
  wait();
  c.write(3);
  wait();
  c.write(4);
  wait();
  c.write(5);
  wait();

 }
```

Simulation: main

- To have a working simulation we need a file with the function sc_main(int, char *arg[]).
- The sc_main() function has to:
  - Declare the channels linking the various modules in the system;
  - Instantiate the modules of the system;
  - Launch the simulation by the sc_start() function;
    - sc_start(simulation_time).

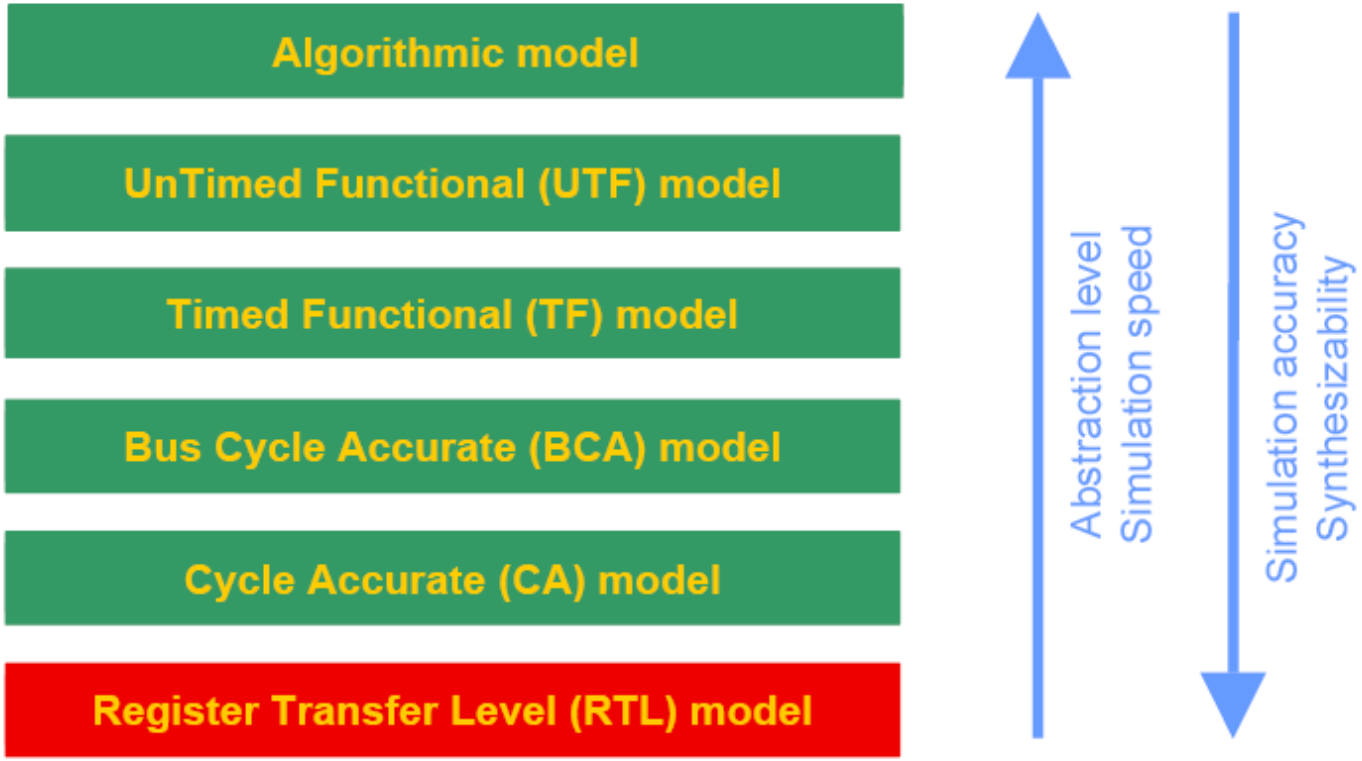
Simulation: example (1/2)

```
#include "systemc.h"
#include "due_moduli.h"
#include "tbench.h"

int sc_main(int, char *v[])
{ /* Dichiarazione dei channel */
  sc_signal<int> a;
  sc_signal<int> b;
  sc_signal<int> c;
  sc_clock clk("Clock", 10,0.5, 0.0);
  sc_signal<int> ris;
 /* Istanziazione dei moduli */
 tbench tb("TB"); /* Istanziazione di tbench*/
 tb(clk,a,b,c);

 due_moduli dm("DueMod"); /* Istanziazione di due_moduli */
 dm(a,b,c,clk,ris);
```

Simulation: example (2/2)

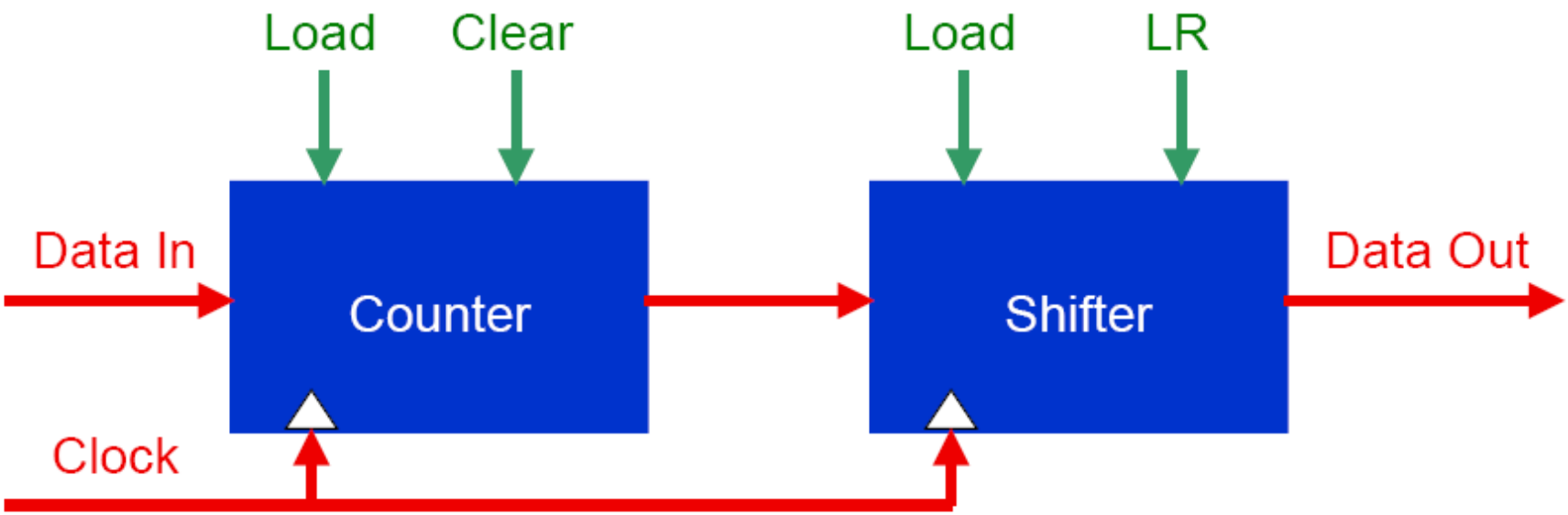
```
  /* creazione ed apertura del file di trace vcd */
  sc_trace_file *tf= sc_create_vcd_trace_file("dueproc");
  /* dichiarazione degli oggetti da visualizzare */
  sc_trace(tf,clk,"CLK");
  sc_trace(tf,a,"A");
  sc_trace(tf,b,"B");
  sc_trace(tf,c,"C");
  sc_trace(tf,ris,"Ris");
  /* avvio della simulazione */
  sc_start(sc_time(160,SC_NS));
  /* chiusura del trace file vcd */
  sc_close_vcd_trace_file(tf);

return 0;
}
```

Layers of Hardware Design



An RTL Model Example

- RTL level: signal accurate, cycle accurate, resource accurate
- Can not use abstractions (functional units, communication infrastructures, …)



An RTL Model Example

- An 8 bit counter. This counter can be loaded on a `clk` rising edge by setting the input `load` to 1 and placing a value on input `din`. The counter can be cleared by setting input `clear` high.
- A very basic 8 bit shifter. The shifter can be loaded on a `clk` rising edge by placing a value on input `din` and setting input `load` to 1. The shifter will shift the data left or right depending on the value of input `LR`. If `LR` is low the shifter will shift right by one bit, otherwise left by one bit.
- Local temporary values are needed because the value of output ports cannot be read.

An RTL Model Example

```
// counter.h
SC_MODULE(counter) {
  sc_in<bool> clk;
  sc_in<bool> load;
  sc_in<bool> clear;
  sc_in<sc_uint<8> > din;
  sc_out<sc_uint<8> > dout;
  unsigned int countval;
  void counting();
  SC_CTOR(counter) {
    SC_METHOD(counting);
    sensitive << clk.pos();
  }
};
```

An RTL Model Example

```
// counter.cpp
#include "counter.h"

void counter::counting()
{
  if (clear)
    countval = 0;
  else if (load.read())
          countval = (unsigned int)din.read();
        else
          countval++;
  dout.write((sc_uint<8>)countval);
}
```

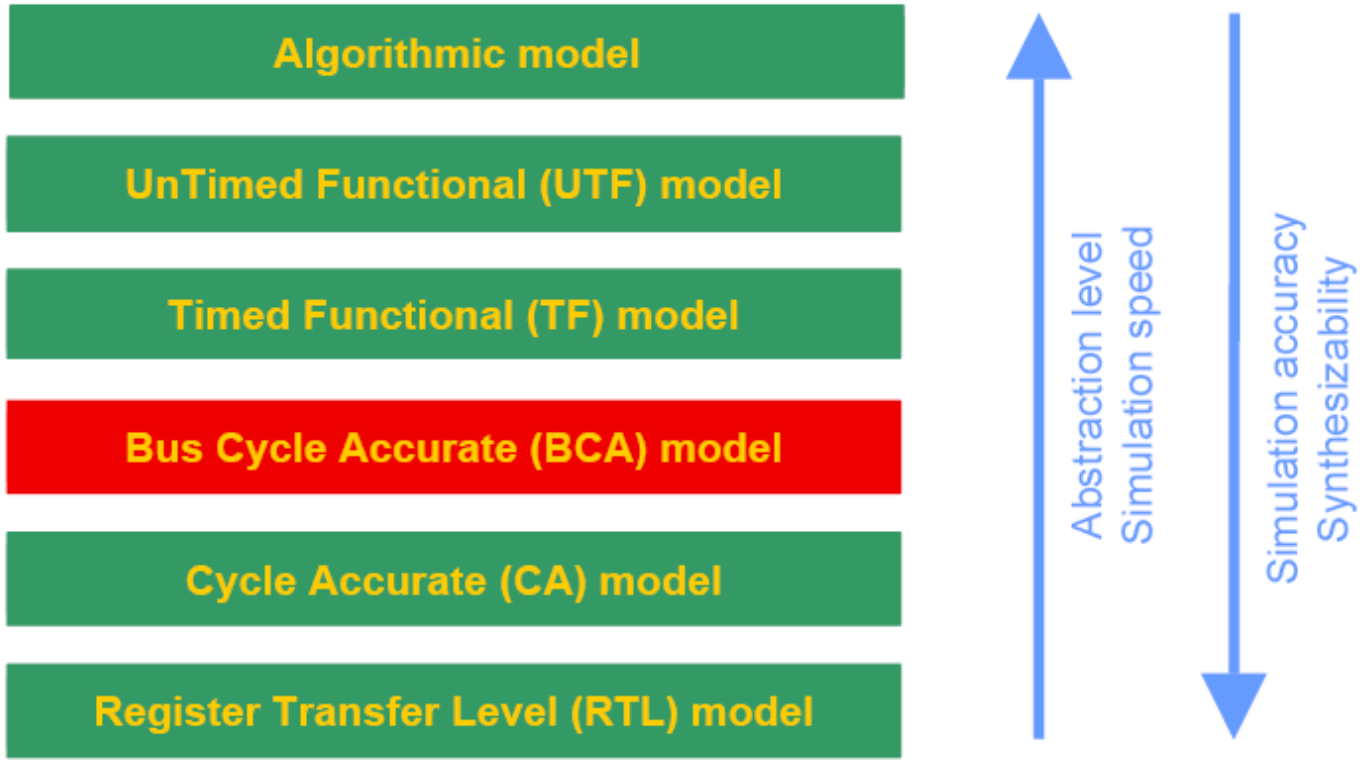
An RTL Model Example

```
// shifter.h
SC_MODULE(shifter) {
  sc_in<sc_uint<8> > din;
  sc_in<bool> clk;
  sc_in<bool> load;
  sc_in<bool> LR;              // shift left if true
  sc_out<sc_uint<8> > dout;
  sc_uint<8> shiftval;
  void shifting();
  SC_CTOR(shifter) {
    SC_METHOD(shifting);
    sensitive << clk.pos();
  }
};
```

An RTL Model Example

```
// shifter.cpp
#include "shifter.h"
void shifter::shifting() {
  if (load.read())
    shiftval = din.read();
  else if (!LR.read()) {              // shift right
        shiftval.range(6,0) = shiftval.range(7,1);
        shiftval[7] = '0'; }
      else if (LR.read()) {           // shift left
            shiftval.range(7,1)=shiftval.range(6,0);
            shiftval[0] = '0'; }
  dout.write(shiftval);
}
```

Layers of Hardware Design



A Bus Cycle Accurate Model Example

- Pin-accurate like RTL, but not cycle-accurate
- Does not imply mapping of computation onto HW resources

- Euclid's algorithm to find the Greatest Common Divisor (GCD) of two numbers:
  - Given a, b, with $a \geq 0$, $b > 0$,
  - If b divides a, then $GCD(a, b) = b$;
  - Else, $GCD(a, b) = GCD(b, a \bmod b)$.

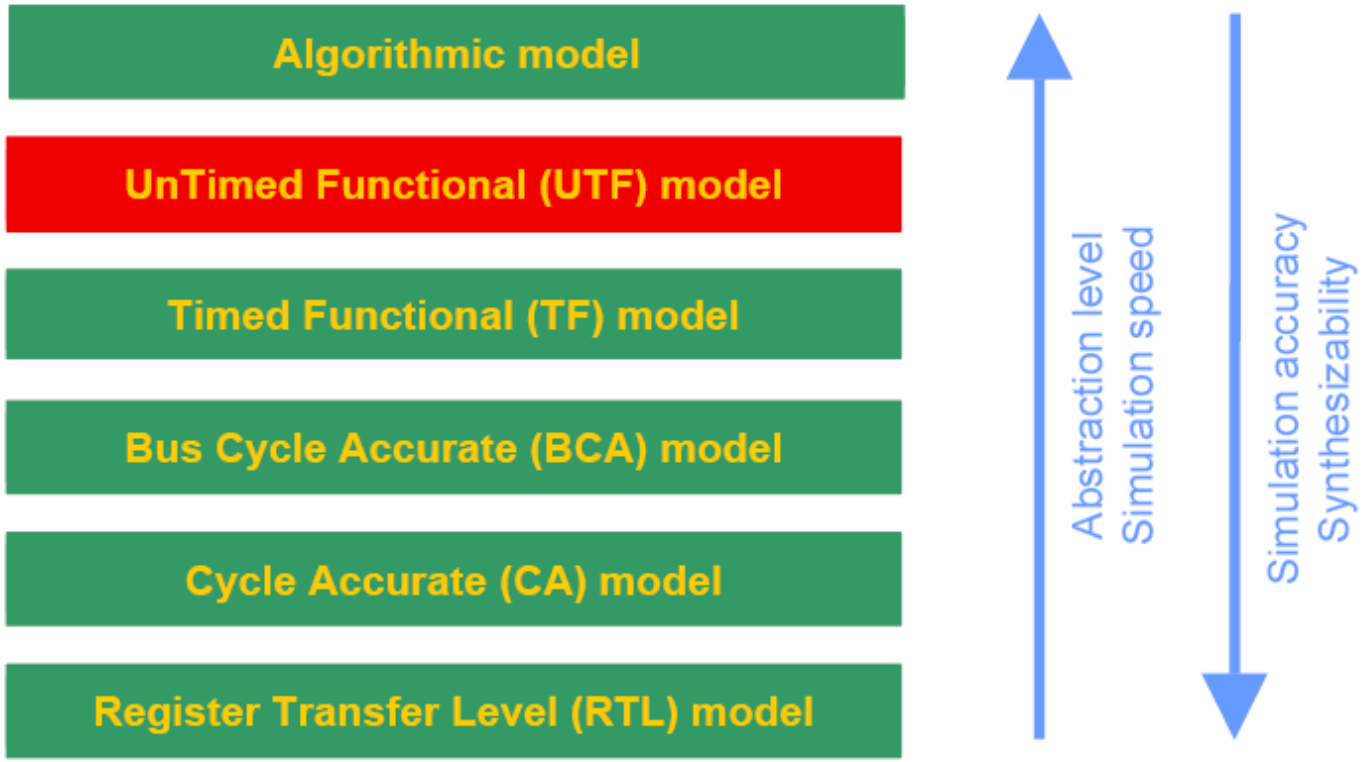
A Bus Cycle Accurate Model Example

```
// euclid.h
SC_MODULE (euclid) {
  sc_in_clk clock;
  sc_in<bool> reset;
  sc_in<unsigned int> a, b;
  sc_out<unsigned int> c;
  sc_out<bool> ready;
  void compute();

  SC_CTOR(euclid) {
      SC_CTHREAD(compute, clock.pos());
      watching(reset.delayed() == true);
  }
};
```

A Bus Cycle Accurate Model Example

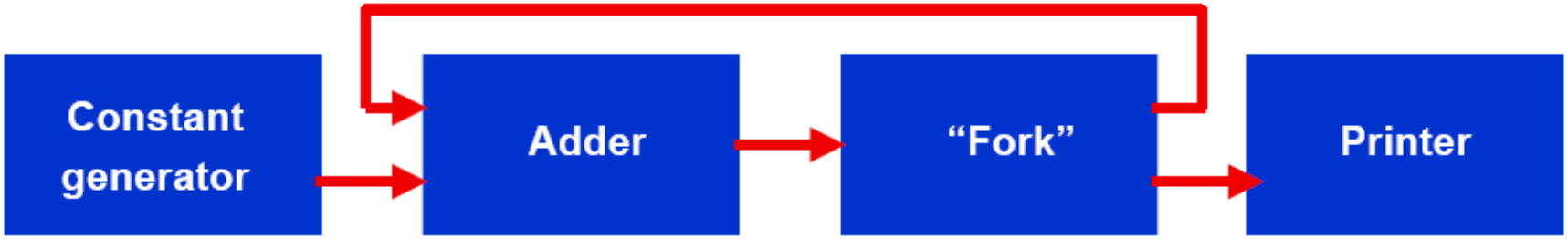
```
// euclid.cpp
void euclid::compute()
{
    unsigned int tmp_a = 0, tmp_b;                    // reset section
    while (true) {
        c.write(tmp_a);                               // signaling output
        ready.write(true);
        wait();                                       // moving to next cycle
        tmp_a = a.read();                             // sampling input
        tmp_b = b.read();
        ready.write(false);
        wait();                                       // moving to next cycle
        while (tmp_b != 0) {                          // computing
            unsigned int r = tmp_a;
            tmp_a  = tmp_b;
            r = r % tmp_b;
            tmp_b = r;
        }
    }
}
```

Layers of Hardware Design



An UnTimed Functional (UTF) Model Example

- Very widespread modeling level
- Describes functionality, but not timing
- Most general form of UTF model: “Kahn Process Networks” (KPN)
- But often implemented as “Dataflow model”: modules communicating with each other via blocking FIFOs



An UnTimed Functional(UTF) Model Example

```
// constgen.h
SC_MODULE(constgen) {
{
   sc_fifo_out<float> output;

   SC_CTOR(constgen) {
        SC_THREAD(generating());
   }

   void generating() {
        while (true) {
                output.write(0.7);
        }
   }
}
```

An UnTimed Functional (UTF) Model Example

```
// adder.h
SC_MODULE(adder) {
{
   sc_fifo_in<float> input1, input2;
   sc_fifo_out<float> output;

   SC_CTOR(adder) {
        SC_THREAD(adding());
   }

   void adding() {
        while (true) {
                output.write(input1.read() + input2.read());
        }
   }
}
```

An UnTimed Functional (UTF) Model Example

```
// forker.h
SC_MODULE(forker) {
{
   sc_fifo_in<float> input;
   sc_fifo_out<float> output1, output2;

   SC_CTOR(forker) {
        SC_THREAD(forking());
   }

   void forking() {
        while (true) {
                float value = input.read();
                output1.write(value);
                output2.write(value);
        }
   }
}
```

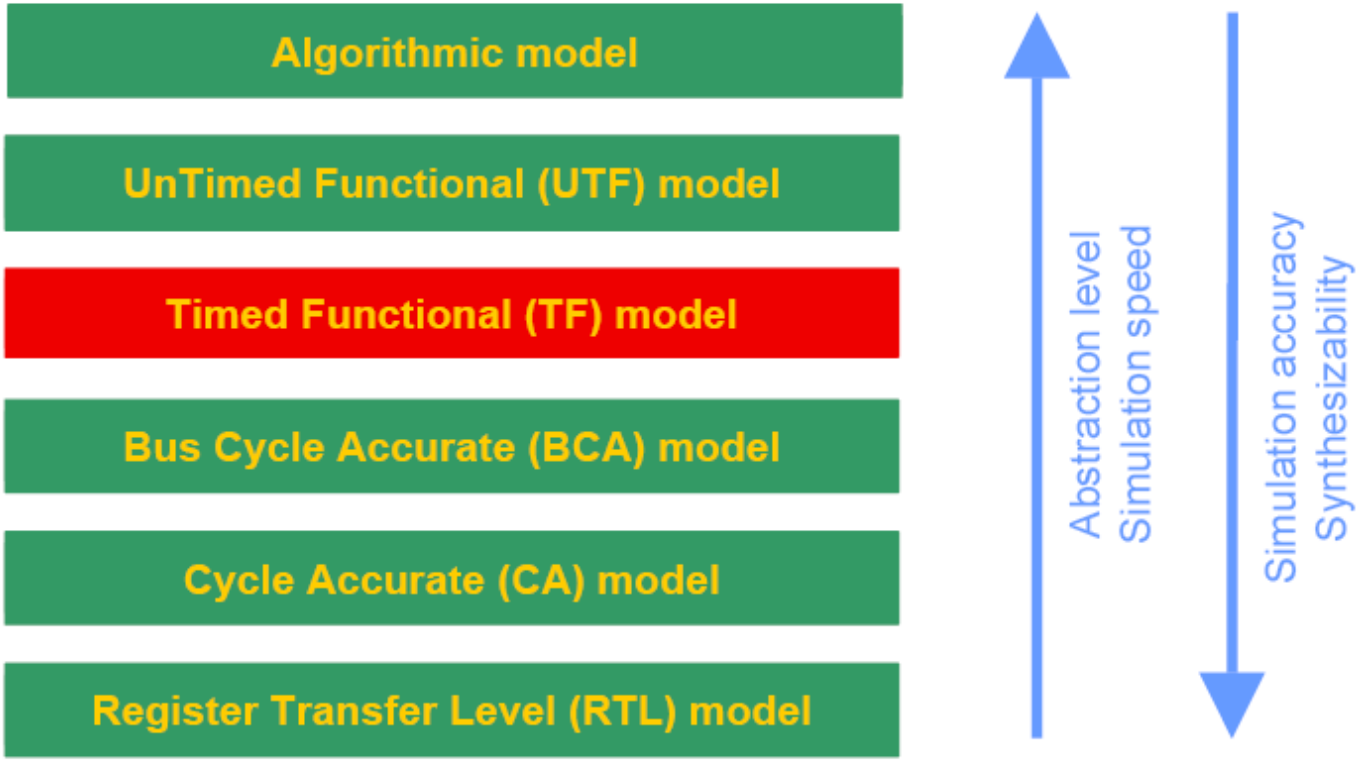
An UnTimed Functional (UTF) Model Example

```
// printer.h
SC_MODULE(printer) {
{
   sc_fifo_in<float> input;

   SC_CTOR(printer) {
        SC_THREAD(printing());
   }

   void printing() {
        for (unsigned int i = 0; i < 100; i++) {
                float value = input.read();
                printf("%f\n", value);
        }
        return;                 // this indirectly stops the simulation
                                // (no data will be flowing any more)
   }
}
```

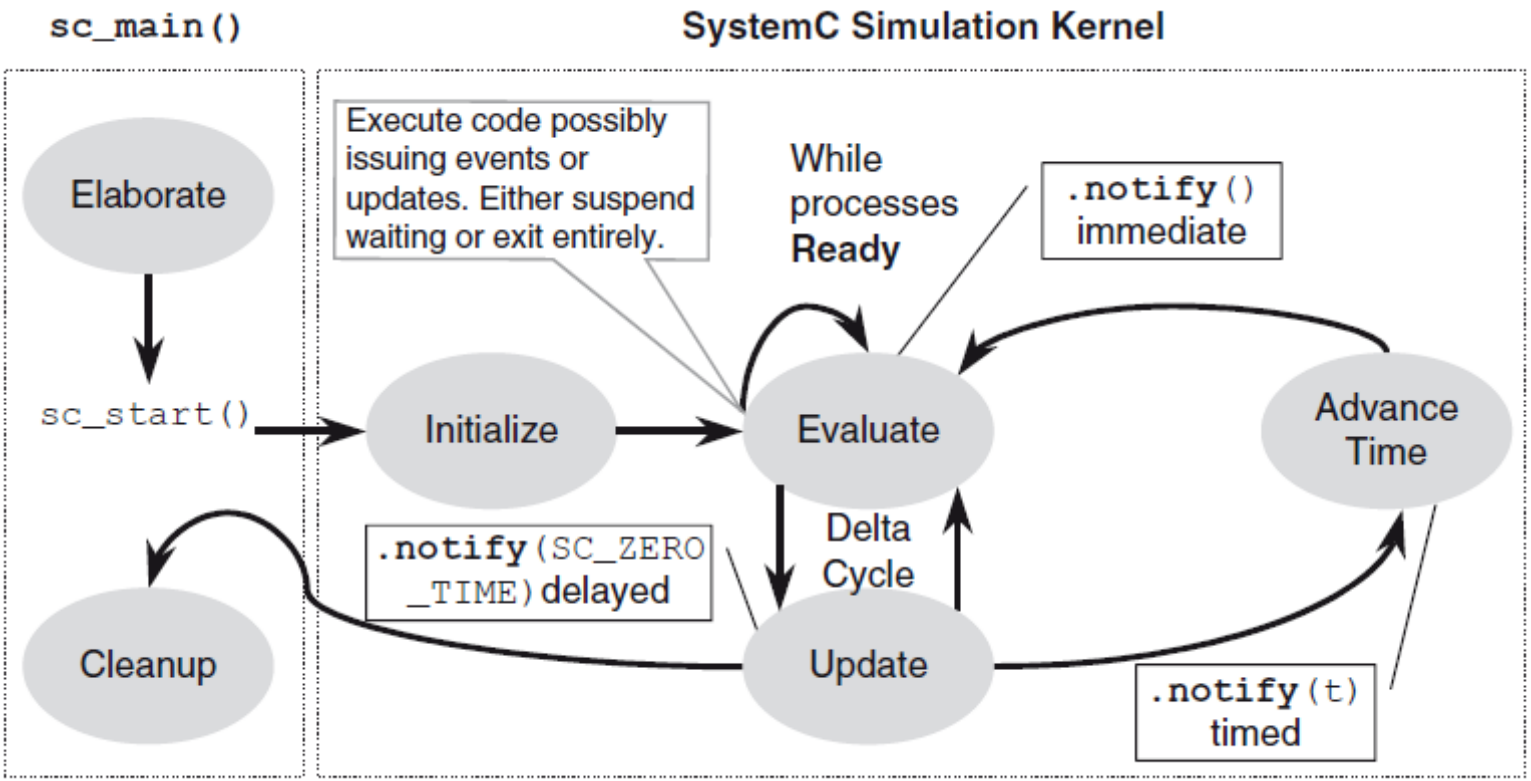
An UnTimed Functional (UTF) Model Example

```
// main.cpp
int sc_main(int, char**) {
   constgen my_constgen("my_constgen_name");                    // module
   adder my_adder("my_adder_name");                             // instantiation
   forker my_forker("my_forker_name");
   printer my_printer("my_printer_name");
   sc_fifo<float> constgen_adder("constgen_adder", 5);          // FIFO
   sc_fifo<float> adder_fork("adder_fork", 1);                  // instantiation
   sc_fifo<float> fork_adder("fork_adder", 1);
   sc_fifo<float> fork_printer("fork_printer", 1);
   fork_adder.write(2.3);                                       // initial setup
   my_constgen.output(constgen_adder); my_adder.input1(constgen_adder);
   my_adder.input2(fork_adder); my_adder.output(adder_fork);
   my_fork.input(adder_fork); my_fork.output1(fork_adder);  // binding
   my_fork.output2(fork_printer); my_printer.input(fork_printer);
   sc_start(-1);                        // simulate "forever". Will exit
   return 0;                            // when no events are queued
}                                       // (printer exits, fifos saturate)
```

Layers of Hardware Design



Refining to Timed Functional(TF) Model

```
// constgen.h
SC_MODULE(constgen) {
{
   sc_fifo_out<float> output;

   SC_CTOR(constgen) {
        SC_THREAD(generating());
   }

   void generating() {
        while (true) {
                wait(200, SC_NS);
                output.write(0.7);
        }
   }
}
```

SystemC simulation kernel